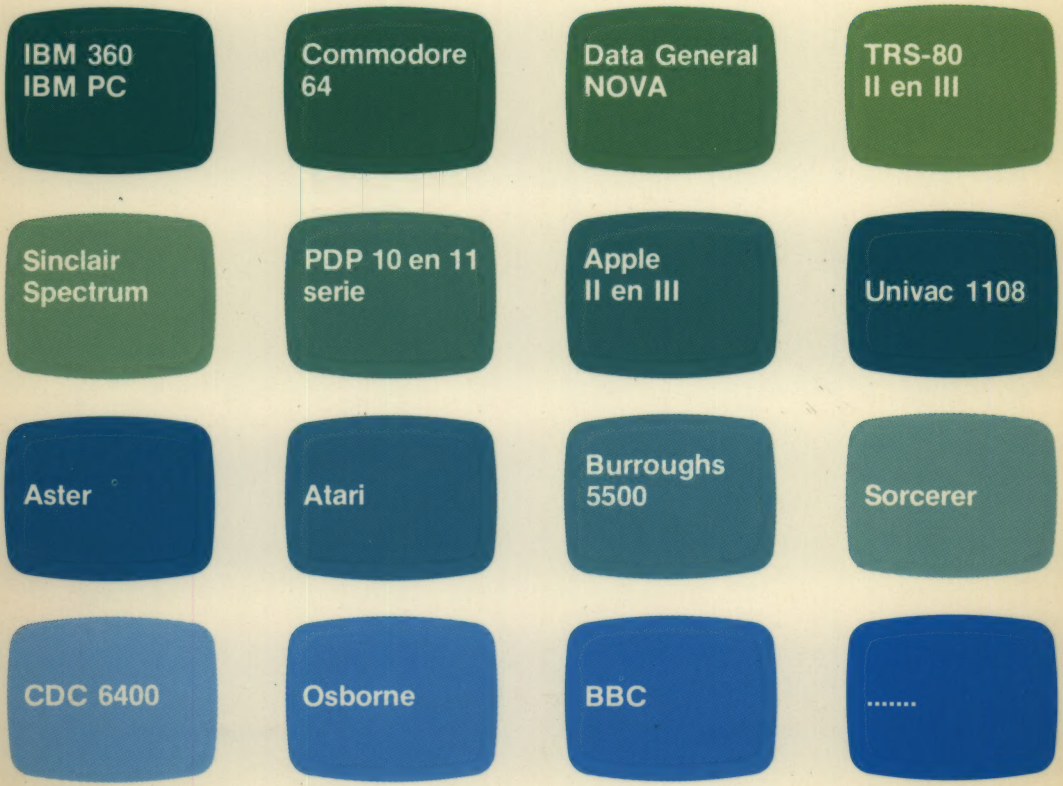
ALAN WINFIELD

flitsend forth

ACADEMIC SERVICE



flitsend forth

ALAN WINFIELD

flitsend forth

ACADEMIC SERVICE

Oorspronkelijke titel:
The Complete FORTH. A new way to program microcomputers
Published by Sigma Technical Press, UK, Wilmslow, Chesshire, 1983

CIP-GEGEVENS

Winfield, Alan

Flitsend FORTH / Alan Winfield ; [vert. uit het Engels]. -
Den Haag : Academic Service
Vert. van: The complete FORTH : a new way to program microcomputers. - Wilmslow : Sigma Technical Press, 1983. - Met lit. opg.
ISBN 90-6233-115-7
SISO 365.3 SVS 8.12.3 UDC 681.31:800.92 Forth
Trefw. : microcomputers / programmeertalen.

Uitgegeven door: Academic Service
Postbus 96996
2509 JJ Den Haag
Omslagontwerp: JAM Gauw
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam
ISBN 90 6233 115 7

VOORWOORD

FORTH is een opmerkelijke programmeertaal, die pas in het begin van de jaren zeventig werd ontwikkeld voor wetenschappelijke toepassingen en nog niet lang geleden op zeer grote schaal voor microcomputers ter beschikking is gekomen. FORTH is inderdaad opgedoken op het moment dat microcomputers 'volwassen' zijn geworden; veel gebruikers hebben zo het experimenteerstadium kunnen overslaan en schrijven programma's voor serieuze en minder eenvoudige toepassingen. Veel bestaande oudere talen kennen ernstige beperkingen. BASIC is te langzaam voor veel toepassingen, terwijl een assemblertaal niet prettig is voor een gebruiker, moeilijk is te leren, en nog erger, beperkt is tot één type machine. FORTH heeft geen last van deze problemen en verschaft een compacte en prettige taal met een snelle verwerking van de programma's.

Dit boek is een complete handleiding voor het programmeren in FORTH. In de eerste helft wordt de taal geïntroduceerd met behulp van veel voorbeelden en vaak ook een vergelijking met BASIC. In de hoofdstukken daarna wordt ingegaan op enkele ongewone eigenschappen van FORTH, waarvan vele geen equivalent hebben in andere talen. De FORTH-79 norm wordt overal in dit boek gebruikt, hoewel enkele oudere nogal voorkomende afwijkingen van die norm worden uiteengezet in de vorm van opmerkingen.

Dit boek is bestemd voor een ieder, die FORTH wil leren en gebruiken. Enige kennis van microcomputers, het gebruik van beeldscherm stations en BASIC wordt vooröndersteld, maar niet een eerdere kennismaking met FORTH. Het boek kan evengoed dienen als een nuttig naslagwerk voor programmeurs die geregeld FORTH gebruiken.

Ik ben erkentelijkheid verschuldigd aan Charles Moore en Elizabeth Rather, die de taal bedachten, en aan de FORTH Standaard Commissie, die de hier gebruikte norm heeft ontwikkeld. Ik bedank verder Graham Beech van de Sigma Technical Press voor de suggestie om dit boek te schrijven, Ian Mitchell en Peter Cain voor hun hulp bij het bewerken van het manuscript, en mijn wederhelft Mary voor haar geduld tijdens het schrijven van dit boek!

Alan Winfield,
Hull, januari 1983

INHOUDSOPGAVE

INLEIDING TOT FORTH

Een van de eerste dingen die nieuwkomers in de informatica ontdekken is dat computers geprogrammeerd moeten worden voordat ze nuttig zijn. Voor beginners met microcomputers betekent dit meestal programmeren in BASIC, maar ervaren computergebruikers en avontuurlijke hobbyisten zullen mogelijk werken met andere talen, zoals PASCAL, FORTRAN en ASSEMBLER (een machinetaal) om er maar enkele te noemen. FORTH is een tamelijk recente toevoeging aan het spectrum van programmeertalen; het biedt een vrij unieke manier van programmeren die helemaal niet lijkt op programmeren in BASIC of een van de andere genoemde talen.

Waarin is FORTH anders?

Forth is vooral anders dan oudere programmeertalen door het feit dat je door het schrijven van programma's de taal uitbreidt zoals we met het volgende voorbeeld zullen laten zien.

Een nuttig programma voor je computer zou 'kalender' kunnen heten omdat het zulke dingen berekent als de weekdag voor een bepaalde datum, of het aantal resterende dagen in het lopende verenigingsjaar, of zelfs een kalender afdrukt voor een gegeven maand en jaar. Zo'n programma zou gemakkelijk in bijvoorbeeld BASIC geschreven kunnen worden, maar voor gebruik zou het steeds vanuit een schijf of cassetteband geladen moeten worden. Eerst zou dan waarschijnlijk een keuzelijst verschijnen waaruit de gebruiker een bepaalde gewenste taak zou moeten selecteren. Kortom, het vergt nogal wat werk om het programma te laden en te verwerken.

De FORTH programmeur zou het probleem op een heel andere manier aanpakken. Hij beslist eerst wat de prettigste manier is om de gewenste informatie op te vragen. Voor het vinden van de weekdag voor een bepaalde datum zal hij bijvoorbeeld kiezen het intikken van een datum, gevolgd door het woord 'dag':

```
1 januari 1982 dag
```

Als alles goed gaat moet FORTH meteen reageren met het antwoord:

Vrijdag

Voor het bepalen van de resterende dagen van een verenigingsjaar zouden we de datum kunnen kiezen, gevolgd door 'resterend':

```
1 december 1981 resterend 30
```

waarbij FORTH dus gereageerd heeft met het antwoord '30'. Om een kalender voor een maand te krijgen tikken we bijvoorbeeld:

```
februari 1982 maand
```

en verwachten dat FORTH zal reageren met het afdrukken van:

```
februari   :1982
  Z  M  D  W  D  V  Z
     1  2  3  4  5  6
  7  8  9 10 ... enz.
```

We zouden zelfs een kalender voor het hele jaar willen krijgen door enkel in te tikken:

```
1982 jaar
```

Na op deze manier het gewenste eindresultaat vastgelegd te hebben, rest ons een FORTH-programma te schrijven voor ieder van de functies 'dag', 'resterend', 'maand' en 'jaar'. Het 'jaar'-programma zal mogelijk de volgende gedaante hebben:

```
: jaar
      12 0 DO                (lus voor 12 maanden)
             I OVER maand    (druk een maand af)
           LOOP
      DROP
;
```

Dit programma, genaamd 'jaar', heeft een speciale structuur die in FORTH een 'Colon' (d.i. het Engelse woord voor dubbele punt)-definitie genoemd wordt. Mits FORTH al het woord 'maand' kent, kan het bovenstaande zonder meer ingetikt worden; het wordt dan vertaald en aan FORTH toegevoegd met de naam 'jaar'. Daarmee is het 'jaar' programma een deel van FORTH geworden, dat op ieder moment uitgevoerd kan worden door bijvoorbeeld alleen maar in te tikken:

```
1982 jaar
```

Kan het eenvoudiger?!

Voordat we 'jaar' intikken, moeten we natuurlijk al een programma voor 'maand' hebben toegevoegd. Wederom zal dit de structuur van een Colon-definitie hebben:

```
: maand        ... een FORTH programma ...    ;
```

In tegenstelling tot het stukje 'jaar'-programma zullen de opdrachten in dit 'maand'-programma waarschijnlijk niet weer naar een ander 'stukje'-programma verwijzen maar direct in basis FORTH-opdrachten geformuleerd kunnen worden.

Maak je niet ongerust wanneer je de precieze regels van de net gegeven voorbeelden niet begrijpt of woorden als 'vertalen' niet kent. Ze zullen in de loop van dit boek verklaard worden, met inbegrip van meer gedetailleerde programma's voor de ingevoerde 'kalender'-woorden. Wat je uit deze inleiding vooral moet onthouden is dat de FORTH programmeur een eigen 'woordenlijst' van stukjes programma opbouwt (een kalender woordenlijst in het voorgaande voorbeeld). Door vervolgens een van de woorden uit de woordenlijst eenvoudigweg in te tikken wordt dan het corresponderende programma verwerkt. Een complete woordenlijst kan op schijf of cassetteband worden opgeslagen.

Voor zowel de hobbyist als de beroepsprogrammeur is dit een interessante en verfrissende aanpak van het programmeren. Voor de beroepsprogrammeur volgen nog enkele mededelingen aan het eind van deze inleiding, maar nu eerst iets over dit boek.

Hoe moet je dit boek lezen?

Zoals bij iedere nieuwe programmeertaal moet je FORTH leren, beginnend op de begane grond. Er zijn heel wat verdiepingen in het FORTH-gebouw, maar zoals hopelijk uit deze inleiding blijkt zal het de moeite lonen om FORTH te leren. FORTH is een interactieve taal, hetgeen betekent dat programma's, aan het toetsenbord zittend, ontwikkeld en getest kunnen worden. Nuttiger in dit stadium is dat we FORTH ook kunnen léren terwijl we aan het toetsenbord zitten. Dit betekent dat de kennismaking met nieuwe mogelijkheden gepaard kan gaan met het beproeven van die mogelijkheden voor een microcomputer waarvoor FORTH beschikbaar is. Dit geldt al meteen vanaf het begin!

Dit boek is een ideale begeleiding voor een gloednieuw FORTH systeem voor je microcomputer, maar maak je geen zorgen als je geen microcomputer bij de hand hebt; de voorbeelden zijn ook dan te begrijpen. Eigenlijk alles van FORTH in de tekst kan ingetikt worden en wordt geaccepteerd door de meeste standaard FORTH systemen die op dit moment beschikbaar zijn. Als jouw systeem in overeenstemming is met de FORTH-79 norm (opgesteld door de FORTH Standaard commissie), dan zullen alle voorbeelden zonder wijziging verwerkt kunnen worden. Is dat niet het geval, dan zul je de documentatie van je systeem moeten raadplegen om mogelijke verschillen te onderkennen.

In alle voorbeelden, geschikt om beproefd te worden met een FORTH systeem, is het antwoord van FORTH cursief aangegeven. Afgezien daarvan zijn er nog twee zaken die je goed moet onthouden:

i) FORTH doet niets met wat je ingetikt hebt (verderop vaak sliert genoemd) totdat je de toets hebt ingedrukt waar 'return' op staat. Deze toets zit helemaal aan de rechterkant van je toetsenbord en in plaats van 'return' zal er soms 'enter' of een pijl naar links op staan. Het plezierige van deze uitgestelde reactie is dat je mogelijke tikfouten kunt corrigeren met behulp van de 'backspace' (d.i. correctie) toets!

ii) FORTH vereist dat ieder getal, woord of symbool (dat soms echter bestaat uit een aantal direct op elkaar volgende toetsaanslagen) gevolgd wordt door tenminste één spatie. De verklaring hiervoor volgt later.

Deze afspraken worden niet steeds herhaald in de voorbeelden, maar gelden altijd zodat bijvoorbeeld:

```
." IK BEN FORTH "   IK BEN FORTH ok
```

betekent dat je ingetikt had `." IK BEN FORTH "`, gevolgd door het aanslaan van de 'return'-toets. FORTH reageerde hierop met het afdrukken van *IK BEN FORTH ok*. Hierbij wordt met 'ok' in FORTH bedoeld: "Ik ben klaar met het verwerken van een invoersliert en verwacht nu de volgende sliert".

Hoofdstukken 1 t/m 5 vormen een op zichzelf staande inleidende cursus voor het programmeren in FORTH, die slechts een elementaire kennis vergt van informatica begrippen. Voor lezers die vertrouwd zijn met BASIC, volgen naast elkaar voorbeelden in FORTH en BASIC als dit zinvol (en mogelijk!) is. Aan het eind van ieder hoofdstuk zijn ook wat oefeningen opgenomen, waarvan de uitwerking achter in het boek is te vinden.

Hoofdstuk 6 gaat over tekstbehandeling in FORTH en over het werken met schijf of cassette. Daar tekstbehandeling niet in alle FORTH systemen op dezelfde manier gebeurt, geeft dit hoofdstuk slechts enkele karakteristieken van dit onderwerp.

Hoofdstukken 7 t/m 9 gaan over een selectie van enkele enigszins buitennissige en soms duistere technieken bij het programmeren in FORTH. Deze hoofdstukken zijn niet essentieel voor de echte beginner met FORTH, maar zijn eerder bestemd voor naslag door actieve FORTH programmeurs, die hopelijk in deze hoofdstukken zullen duiken om tips en ideeën op te doen voor hun eigen programma's. Nieuwkomers met FORTH wordt niettemin aangeraden vluchtig door deze hoofdstukken heen te gaan om de smaak te pakken te krijgen en zich bewust te worden van enkele ongewone eigenschappen van FORTH. Veel van deze eigenschappen vinden we niet bij de meeste andere programmeertalen.

Hoofdstuk 10 geeft tenslotte twee grotere FORTH programma's, die zelf interessant zijn en laten zien hoe een FORTH programmeur het ontwerpen van grote programma's aanpakt.

Een handig overzicht

In dit boek vind je een losse overzichtskaart, die in het kort details geeft van alle woorden en symbolen, die FORTH kent. Samen heten deze woorden en symbolen in de FORTH terminologie het "Woordenboek". Als je bij je microcomputer een FORTH systeem hebt, zal het woordenboek daarvan waarschijnlijk niet precies gelijk zijn aan dat van de overzichtskaart, maar de verschillen zijn maar klein. De op de overzichtskaart gebruikte notatie om het effect van ieder FORTH woord of symbool kort te beschrijven, zal in het begin wat verwarrend zijn, maar het wordt uitgelegd in het eerste hoofdstuk. Zodra je je eigen FORTH programma's gaat schrijven, wat niet lang zal duren, zul je je de waarde van deze overzichtskaart realiseren. Aanbevolen wordt deze steeds te raadplegen wanneer je voorbeelden gaat uitwerken, liefst zo gauw mogelijk na dat eerste hoofdstuk. Wanneer je opheldering behoeft t.a.v. de FORTH terminologie kun je die evenzo vinden in de verklarende woordenlijst achter in dit boek; dit zal eenvoudiger zijn dan het zoeken van de desbetreffende tekst in de hoofdstukken.

De rest van deze inleiding geeft een samenvatting van FORTH voor beroepsprogrammeurs. Wanneer je nog niet tot die klasse behoort, kun je deze paragraaf overslaan en meteen overstappen naar het eerste hoofdstuk.

Voor beroepsprogrammeurs

In alle opzichten is FORTH een hoogst ongewone programmeertaal. Het heeft weinig gemeen met veel gebruikte talen als BASIC of PASCAL. Niettemin is FORTH een 'hogere' taal; concepten voor gestructureerde programmering zijn er in opgenomen en FORTH programma's zijn zowel modulair als overdraagbaar. Evengoed kan de FORTH programmeur beschikken over dicht bij de machine staande instructies of over die uit een symbolische assembleertaal, zodat in sommige opzichten FORTH vergeleken kan worden met een macro-assembleertaal.

Een FORTH-systeem is zowel een vertolker als een vertaler. Als regel wordt alle invoer voor FORTH (die afkomstig kan zijn uit het toetsenbord of een achtergrondgeheugen) geïnterpreteerd en direct verwerkt. Wanneer deze invoer echter opgenomen is in een Colon-definitie (zoals eerder vermeld in deze inleiding), dan wordt die vertaald in een compact machinetaalprogramma. FORTH heeft dus de ongebruikelijke eigenschap voor het testen en corrigeren van programma's om zich te gedragen als een interactieve vertolker, die prettig en gemakkelijk is te gebruiken, terwijl de uiteindelijke programma's echt vertaald worden en daardoor snel en efficiënt zijn.

De snelheid van verwerking is ongeveer die van vertaalde PASCAL programma's of ruim tien maal sneller dan vertolkte BASIC programma's.

Een ander kenmerk van FORTH is de ongebruikelijke manier van programmeren door uitbreiding van de taal, hetgeen resulteert in snelle ontwikkeling en correctie van programma's. Alle FORTH bewerkingen (die vergeleken kunnen worden met de 'gereserveerde' woorden van BASIC als 'LET', 'PRINT', '+','-', enz.) zijn opgenomen in een 'Woordenboek'. Het programmeren gebeurt door nieuwe woorden te definiëren en deze nieuwe woorden aan het Woordenboek toe te voegen met behulp van de speciale Colon-definitie. Deze nieuwe woorden kunnen op hun beurt gebruikt worden om nog ingewikkelder bewerkingen te programmeren. Op deze manier bouwt de FORTH programmeur zijn eigen 'woordenschat' op, die helemaal afgestemd is op zijn probleem. Iedere nieuwe bewerking wordt volledig via het toetsenbord getest alvorens met de volgende bewerking te beginnen en op deze manier worden fouten in een vroeg stadium ontdekt en verbeterd!

De meeste volledige FORTH systemen zullen al van speciale 'Woordenboeken' voorzien zijn, zoals Woordenboeken voor tekstbewerking, toegang tot schijfgeheugens en assemblertaal bewerkingen. Dit betekent dat een FORTH systeem gewoonlijk gebruikt kan worden zonder de hulp in te roepen van andere programmatuur voor het ontwikkelen van programma's. Een heel systeem is verder vrij compact en vergt meestal minder dan 16 Kbytes. Dit is zeer prettig voor degene die programmatuur ontwikkelt omdat hij dan het computersysteem waarop een toepassing verwerkt moet worden tevens kan gebruiken om de programma's te ontwikkelen.

Wanneer deze beschrijving de indruk wekt dat FORTH de wensdromen van alle programmeurs vervult, dan is het nu tijd om enkele kenmerken van FORTH te vermelden, die sommige programmeurs minder prettig zullen vinden...

Allereerst is het gebruik van een stapel te noemen en dientengevolge de postfix notatie voor expressies. Deze kenmerken zijn grotendeels een gevolg van de structuur van FORTH, die zeker bijdraagt tot de snelheid en compactheid van FORTH. De stapel bevordert ook de handige manier van doorgeven van parameters in programma's. In de eerder beschreven 'kalender' bewerkingen tikt een gebruiker voor het verkrijgen van een kalender voor het hele jaar eenvoudig bijvoorbeeld:

```
1982 jaar
```

Wanneer FORTH deze invoerregel interpreteert, wordt het getal 1982 op de stapel gezet (zoals alle getallen in een te interpreteren invoerregel). Het programma 'jaar' plukt dit getal weer van de stapel af en gebruikt het als parameter (als invoerwaarde voor het programma).

Een tweede, mogelijk aanvechtbaar kenmerk is het gebruik van gehele getallen. De hiertoe leidende overweging in FORTH is dat het rekenen met gehele getallen veel sneller gaat dan dat met decimale getallen en bovendien vergen de meeste toepassingen alleen gehele getallen. Voor toepassingen waar decimale getallen gebruikt moeten worden verschaft FORTH overigens bewerkingen voor extra lange (32 bits) gehele getallen, waarmee decimale rekenkundige berekeningen geprogrammeerd kunnen worden.

Ter afsluiting van deze inleiding zij opgemerkt dat FORTH niet gemakkelijk onder de knie te krijgen is. Gek genoeg is FORTH het eenvoudigst voor beginners in de informatica! Voor lezers als de auteur van dit boek, die opgevoed werden met algebraïsche programmeertalen als PASCAL en BASIC, betekent het leren van FORTH het aanleren van een geheel nieuwe en opmerkelijke aanpak van het programmeren.

1 GRONDSLAGEN VAN FORTH

De eenvoudigste toepassing van een FORTH systeem is het gebruik als een rekenmachine voor het uitwerken van een rekenkundige expressie, gevolgd door het afdrukken van het resultaat. Hoewel dit wel een erg bescheiden begin lijkt, laat het iets ongewoons zien, nl. hoe FORTH een stapel gebruikt en dientengevolge de postfix notatie van een expressie. In dit hoofdstuk worden deze twee concepten geïntroduceerd, alsmede een notatie om het stapelgedrag te beschrijven. Deze notatie wordt in de rest van dit boek steeds gebruikt.

1.1 Praat FORTH

Denk je eens in dat je voor een computer zit die FORTH 'praat' en neem bijvoorbeeld aan dat je hulp nodig hebt voor het vermenigvuldigen van de getallen 23 en 34. In FORTH moet je dan tikken:

```
23 34 * .
```

waarop FORTH dan welwillend zal antwoorden

```
782 ok
```

Kennelijk hebben we hetzelfde resultaat verkregen als met het intikken van `PRINT 23 * 34` in BASIC. Je zult echter de afwijkende plaats van het {*} vermenigvuldigingssymbool in de FORTH versie van deze bewerking hebben opgemerkt, namelijk na de twee met elkaar te vermenigvuldigen getallen in plaats van er tussen, zoals we gewend zijn. Wat is verder de betekenis van het {•} puntsymbool in de FORTH invoer? Deze zaken worden duidelijk als we ons realiseren dat FORTH een z.g. 'stapel' gebruikt voor het rekenwerk.

1.2 De stapel

Neem nu eens een eenvoudiger voorbeeld dan het vorige en wel het intikken van een enkel getal (we herhalen nog een keer wat in de inleiding gezegd is, nl. dat de invoersliert gevolgd wordt door het indrukken van de return-toets):

```
27
```

FORTH zal reageren met het antwoord 'ok' op dezelfde regel:

```
27 ok
```

en het lijkt wel of er niets gebeurd is (behalve dat FORTH schijnt te denken dat alles 'ok' is!). Maar in werkelijkheid is iets belangrijks gebeurd, namelijk dat het getal '27' op de stapel geplaatst is. Het puntsymbool {•}, dat we eerder tegenkwamen, heeft precies het omgekeerde effect en wel dat het bovenste getal van de stapel afgehaald wordt en wordt afgedrukt. Daarom zal de invoer

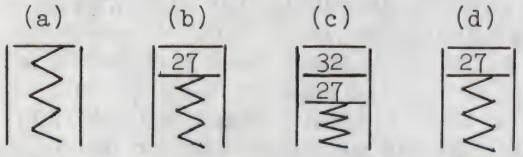
bewerkstelligen dat FORTH afdrukt:

```
27 ok
```

(merk op dat het getal 27 nu cursief is gedrukt!). De stapel onthield dus voor ons een getal, net zoals onze notitie in een agenda.

Laten we de stapel en de operaties van iets op de stapel plaatsen en er weer afhalen in wat meer detail bekijken. Als je hiermee al vertrouwd bent, kun je echter verder gaan met de volgende paragraaf.

Een stapel is een speciaal buffergeheugen voor getallen, die daarin geplaatst worden om ze te bewaren en ze er later weer uit te halen. Het laatst geplaatste getal komt er weer het eerst uit, zodat een stapel ook wel een lifo-geheugen genoemd wordt (naar het Engelse 'last-in-first-out'). Als je je de werking van een stapel wilt voorstellen, moet je maar denken aan een bordenautomaat die wel in kantines of cafetaria's wordt gebruikt. Een veer onder een steunplaat drukt dan de borden omhoog.



Figuur 1.1 De werking van een stapel

Een lege stapel ziet er uit als figuur 1.1 (a). Zet 27 op de stapel en je krijgt figuur 1.1 (b). Er is nog genoeg ruimte in de stapel, zodat we er nog een ander getal op kunnen plaatsen als in (c). Haal je dan een getal van de stapel, dan verdwijnt het getal 32 en je krijgt (d), wat hetzelfde is als (b).

In de volgende twee paragrafen zullen we bekijken hoe het rekenen met behulp van een stapel gebeurt. Weet je dit al en ken je ook al de begrippen infix en postfix notatie, dan kun je doorgaan met paragraaf 1.5.

1.3 Rekenen in FORTH

Voor de eerder behandelde vermenigvuldiging schetsen we nu de inhoud van de stapel voor en na verwerking van ieder getal en symbool in de Forth expressie:

```
23 34 * .
```

Figuur 1.2

Lezen we figuur 1.2 van links naar rechts, dan zien we dat FORTH de getallen van de invoersliert gewoon op de stapel zet. Bij het bereiken van het symbool {*} bevat de stapel dus al de getallen 23 en 34. FORTH reageert op {*} met het verwijderen van de bovenste twee getallen van de stapel, het vermenigvuldigen van die twee getallen en het terugzetten op de stapel van het resultaat 782. Het laatste symbool van de invoersliert {.} verwijdert, zoals al uiteengezet, het bovenste getal van de stapel en drukt het af.

We zijn nu in staat om de eerste programmeringsregel in FORTH te formuleren:

Al het rekenwerk gebeurt in FORTH op de stapel.

Wat vollediger gezegd: alle rekenkundige bewerkingen gebeuren met getallen op de stapel en de resultaten worden teruggeplaatst op die stapel. Dit verklaart nu de ongewone volgorde in:

```
23 34 * .
```

in plaats van het gebruikelijke `PRINT 23 * 34`.

1.4 Meer over rekenen in FORTH

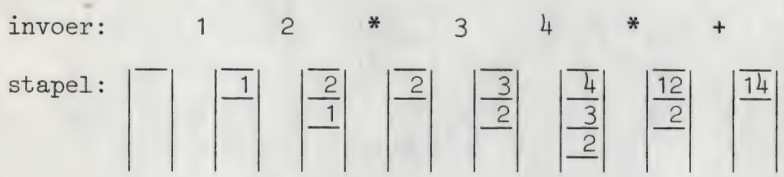
Wat je je nu zult afvragen is: "Betekent dit dat ik voor ingewikkelder rekenwerk een expressie eerst moet omvormen wil het verwerkt kunnen worden met een stapel?". Het antwoord op deze vraag is zonder meer "ja"; je moet een expressie voor verwerking met FORTH eerst veranderen, maar dat is gemakkelijk te leren. Laten we een eenvoudig voorbeeld nemen, nl. de expressie

```
(1 * 2) + (3 * 4)
```

Als we dit uit het hoofd uitrekenen, denken we: "Oh, dat is de som van 1 vermenigvuldigd met 2 (=2) en 3 vermenigvuldigd met 4 (=12). Het antwoord is 2 + 12 = 14". Wat we echt deden was 1 * 2 te berekenen, dit deelresultaat voor later te bewaren, dan 3 * 4 te berekenen, en de twee deelresultaten bij elkaar op te tellen. Laten we dit nu in FORTH opschrijven:

```
1 2 * 3 4 * +
```

Figuur 1.3 laat zien hoe FORTH dit met de stapel uitvoert:



Figuur 1.3

Merk op waar FORTH het resultaat van de eerste vermenigvuldiging bewaart totdat de tweede vermenigvuldiging is uitgevoerd en de optelling kan gebeuren.

De gebruikelijke manier om een expressie op te schrijven, heet de 'infix' notatie omdat daarbij de bewerkingssymbolen (de 'operatoren' +, -, *, /) 'gefixeerd' zijn tussen de getallen (de 'operanden') in. In FORTH moeten we de 'postfix' notatie gebruiken, waarbij de operatoren volgen op de desbetreffende operanden. Zoals we hierboven zagen herschrijf je van infix naar postfix notatie (waarbij de operatoren verhuizen, maar de volgorde van de operanden dezelfde blijft!) door even na te denken hoe je zo'n berekening zelf op papier zou uitvoeren. Als controle kun je dan nog nagaan hoe FORTH die berekening met een stapel zou doen.

Na al dit gepraat over infix en postfix notatie zul je je misschien afvragen waarom je je om FORTH zult bekommeren, temeer omdat de meeste andere programmeertalen als BASIC en PASCAL de infix notatie gebruiken. Het antwoord op die vraag is dat een uitdrukking in de postfix notatie veel sneller verwerkt kan worden dan een infix uitdrukking in BASIC; een PASCAL vertaalprogramma vertaalt een infix uitdrukking om dezelfde reden eerst naar een postfix expressie. Voorts wordt de stapel in FORTH nog voor andere zaken dan het rekenwerk gebruikt, nl. om parameters (invoergetallen en resultaten bijvoorbeeld) door te geven. Het gebruik van de postfix notatie is vrijwel een direct gevolg van het feit dat FORTH een stapel-georiënteerde taal is.

In alle voorgaande voorbeelden waren de getallen steeds gehele getallen, of zoals ze officieel heten: *integers*. Hierop zullen we in de volgende paragraaf verder ingaan.

1.5 Over getallen

In FORTH kennen we geen (in andere talen wel!) 'drijvende-komma' getallen als 3.14 E -2 (wat hetzelfde is als 0.0314). Dit is niet zo'n verschrikkelijke beperking als het op het eerste gezicht lijkt, omdat het in FORTH wel mogelijk is om 'vaste komma' getallen, als 100.23 of 1.234 te gebruiken in bewerkingen met 'dubbele nauwkeurigheid'. Hierop zullen we echter pas in hoofdstuk 8 in detail terugkomen. Nu zullen we ons tot integers beperken.

FORTH kent zowel negatieve als positieve getallen in het bereik

```
- 32768 t/m + 32767
```

zodat -9999, -1, 0 of 10000 'legale' FORTH getallen zijn. Deze 'enkele nauwkeurigheid getallen met teken' worden op de stapel met 16 bits gerepresenteerd (bits zijn nullen of enen; zie voor meer details wat er in de verklarende woordenlijst achter in het boek gezegd wordt over de twee-complement rekenmethode).

Daarnaast maakt FORTH het mogelijk te werken met 'enkele nauwkeurigheid getallen zonder teken' in het bereik

```
0 t/m 65535
```

Dit kan nuttig zijn wanneer we een langer positief bereik, maar geen negatief bereik nodig hebben. Onthoud dan echter dat {•} getallen zonder teken groter dan 32767 niet correct zal afdrukken maar verkeerd, bijvoorbeeld:

```
50000 • -15536 ok
```

Voor een correcte afdruk moeten we dan {U.} gebruiken:

```
50000 U. 50000 ok
```

De meeste rekenbewerkingen in FORTH zullen ook werken voor getallen zonder teken, mits het resultaat binnen het bereik van getallen zonder teken is! Bijvoorbeeld:

```
50000 67 + U. 50067 ok          (d.i. 50000 + 67)
40000  1 - U. 39999 ok          (d.i. 40000 - 1)
20001  3 * U. 60003 ok          (d.i. 20001 * 3)
```

Daar een soortgelijk 'mits' ook geldt voor getallen met teken, is het zaak voorzichtig te zijn wanneer je met grote getallen werkt! Behalve wanneer we uitdrukkelijk praten over getallen 'zonder teken', zullen we verder in dit boek onder 'getal' altijd een 'enkele nauwkeurigheid integer met teken' verstaan.

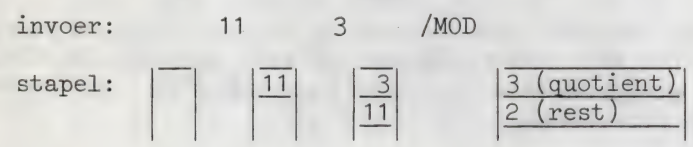
Het gebruik van integers brengt met zich mee dat in FORTH de deling soms niet een correct resultaat zal opleveren. Bijvoorbeeld:

```
11 3 / . 3 ok
```

terwijl het antwoord is 3 met een rest van 2. Om dit resultaat te krijgen moeten we gebruik maken van de FORTH bewerking {/MOD}; een bijzondere deling die zowel het resultaat (quotient) als de rest op de stapel achterlaat. Met de invoersliert

```
11 3 /MOD . . 3 2 ok
```

krijgen we dan het volledige antwoord, nl. 3 met rest 2. Figuur 1.4 laat zien hoe de stapel dit uitvoert:



Figuur 1.4 De {/MOD} bewerking

Als we uitsluitend de rest van een deling willen hebben, moeten we de {MOD} bewerking gebruiken:

```
11 3 MOD . 2 ok          (bereken alleen de rest)
```

Twee minder gebruikelijke rekenbewerkingen zijn {MAX} en{MIN}, die beiden werken met de twee bovenste getallen op de stapel (en die verwijderen). {MAX} laat het grootste van de twee getallen op de stapel achter, {MIN} het kleinste. Bijvoorbeeld:

```
10 20 MAX . 20 ok
-5 5 MIN . -5 ok
```

{MAX} en {MIN} houden dus rekening met het teken van de twee getallen en de afspraak dat negatieve getallen kleiner zijn dan positieve.

FORTH heeft tenslotte de operaties {ABS} en {NEGATE}, die werken op het 'teken' van het bovenste getal op de stapel. De eerste heeft het effect dat het teken van dat getal positief wordt, ongeacht wat het eerst was. De tweede verandert altijd het teken van dat getal. Bijvoorbeeld:

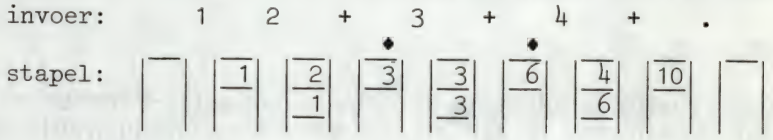
```
100 ABS . 100 ok
-2 ABS . 2 ok
100 NEGATE . -100 ok
-2 NEGATE . 2 ok
```

1.6 Iets over dupliceren

Neem aan dat we vier getallen bij elkaar op willen tellen, maar dat we in plaats van het eindresultaat ook de deelresultaten van iedere stap in de berekening willen hebben. Voor bijvoorbeeld 1 + 2 + 3 + 4 willen we afgedrukt krijgen:

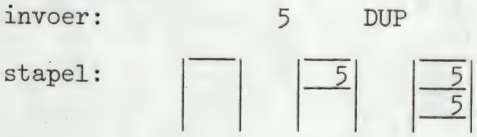
3
6
10

omdat 3 het resultaat is van 1 + 2, 6 dat van 1 + 2 + 3 en 10 het eindresultaat. Wanneer we een invoersliert intikken als in figuur 1.5



Figuur 1.5

krijgen we inderdaad het eindresultaat 10, maar niet de gewenste tussenresultaten. Naar figuur 1.5 kijkend, zien we die tussenresultaten op de met ♦ aangeduide plaatsen, maar we kunnen ze niet afdrukken zonder ze tegelijkertijd van de stapel te verwijderen! Om dit afdrukken te bewerkstelligen gaan we de FORTH operatie {DUP} gebruiken, die niet een rekenkundige bewerking is, maar behoort tot de z.g. stapelmanipulatie bewerkingen. Uitvoering van {DUP} heeft het effect dat een duplicaat van het bovenste getal op de stapel daarboven geplaatst zal worden. Zo zal bijvoorbeeld 5 DUP tot resultaat hebben dat de twee bovenste stapelplaatsen het getal 5 bevatten, zoals in figuur 1.6:



Figuur 1.6

We voeren nu een verkorte notatie in, waarmee we de toestand van de stapel voor en na de uitvoering van een FORTH opdracht kunnen vastleggen:

```
invoer:          DUP

stapelwerking: (n → n n)
```

Aan beide kanten van de pijl (→) is de inhoud van de stapel weergegeven,

waarbij links van de pijl de stapelelementen voor de uitvoering van de opdracht staan en rechts van de pijl die daarna. Het bovenste stapelelement staat in beide gevallen het meest rechts in een rij elementen (waarvan alleen de elementen genoemd worden die betrokken zijn bij de uitvoering van de opdracht).

Terugkomend op het oorspronkelijk gestelde probleem, zal het nu duidelijk zijn dat het afdrukken van het bovenste stapelelement zonder het van de stapel te verwijderen, bereikt kan worden met

```
DUP .
```

Is er slechts één getal op de stapel, dan kan met onze stapelnotatie het effect van {•} beschreven worden met:

```
(n → ) en druk n af
```

Staan er twee of meer getallen op de stapel, dan zal {•} het bovenste stapelelement verwijderen na het afgedrukt te hebben, of wel in onze stapelnotatie:

```
(n2 n1 → n2) en druk n1 af
```

Het gecombineerde resultaat van {DUP} en {•} is dus het ongewijzigd laten van de stapel, hoeveel getallen die ook mag bevatten, samen met het afdrukken van een duplicaat van het bovenste getal:

```
   DUP       .
(n  →  n  n  →  n) en druk n af
```

De oplossing van ons probleem wordt gegeven door:

```
1 2 + DUP . CR 3 + DUP . CR 4 + .
```

hetgeen zal afdrukken:

```
3
6
10 ok
```

omdat we nog een FORTH bewerking hebben ingevoerd, nl. {CR} dat op de schrijfmachine een nieuwe regel plus een wagenterugloop veroorzaakt en daarom ieder resultaat op een nieuwe regel plaatst. Deze bewerking beïnvloedt niet de stapel:

```
 CR
( → )
```

N.B. CR staat voor Carriage Return.

1.7 Nog meer stapel manipulaties

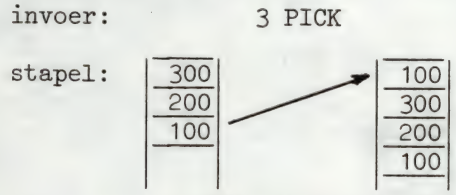
De meeste stapel manipulaties zijn ontworpen om één van de beperkingen van een stapel te overwinnen, nl. dat de volgorde van getallen op een stapel de omgekeerde is van de volgorde van het plaatsen op een stapel. Daarom zal bijvoorbeeld:

```
100 200 300 ok
. . . 300 200 100 ok
```

de ingevoerde getallen in omgekeerde volgorde afdrukken. De stapel manipulatie opdrachten {DUP}, {OVER} en {PICK} maken het mogelijk om een getal op een willekeurige plaats in de stapel te bereiken en een duplicaat daarvan boven op de stapel te plaatsen. Zo zal bijvoorbeeld:

```
100 200 300 ok
3 PICK . 100 ok
```

de uitvoer 100 tot gevolg hebben omdat de opdracht {3 PICK} het derde element uit de stapel 'pikt', een duplicaat boven op de stapel zet en {•} voor het afdrukken zorgt. Figuur 1.7 geeft dit in detail weer. De opdracht {OVER} doet hetzelfde als {2 PICK} en plaatst dus een kopie van het tweede element in de stapel er bovenop.

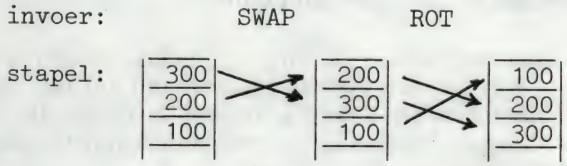


Figuur 1.7 De opdracht {PICK}

De manipulatie opdrachten {SWAP}, {ROT} en {ROLL}, om de volgorde van (ten hoogste drie) getallen boven op de stapel te veranderen, zullen met

```
100 200 300 ok
SWAP ROT ok
. . . 100 200 300 ok
```

in overeenstemming met figuur 1.8 bewerkstelligen:



Figuur 1.8 {SWAP ROT}

dat de volgorde van de getallen op de stapel precies omgedraaid is.

Een algemenere vorm van herrangschikken van getallen op de stapel is mogelijk met de opdracht {n ROLL}, die het effect heeft dat het n-de getal in de stapel 'gerold' wordt naar de top van de stapel (en de getallen daaronder 'een plaats zakken'). Dus {2 ROLL} is hetzelfde als de opdracht {SWAP}, en {3 ROLL} als {ROT}.

Tenslotte moet de opdracht {DROP} genoemd worden, die het effect heeft dat alleen het getal op de top van de stapel wordt verwijderd (zonder dat het afgedrukt wordt zoals met {•} het geval is).

1.8 Samenvatting en oefeningen

Nu volgt een samenvatting van de in dit hoofdstuk behandelde FORTH opdrachten, gevolgd door een aantal oefenopgaven waarvan modeloplossingen achter in het boek staan.

In de beschrijving van de stapeltoestand voor en na uitvoering van die opdrachten betekent n een enkele nauwkeurigheid getal (met teken), en un een enkele nauwkeurigheid getal zonder teken. Denk er ook om dat n1,n2, enz. niet de volgorde van de getallen op de stapel aangeeft; het bovenste getal op de stapel staat steeds rechts in een rijtje getallen voor en na de pijl.

Stapel manipulatie

DUP (n → n n)
Dupliceert het getal op de top van de stapel.

DROP (n →)
Haalt het getal op de top van de stapel weg.

SWAP (n1 n2 → n2 n1)
Verwisselt de bovenste twee getallen op de stapel.

OVER (n1 n2 → n1 n2 n1)
Zet op de stapel een kopie van het tweede getal op de stapel.

ROT (n1 n2 n3 → n2 n3 n1)
'Roteert' de bovenste drie getallen van de stapel, zodat het derde bovenop komt.

n PICK (n1 → n 'n-de getal uit de stapel')
Dupliceert het n-de getal uit de stapel op de top van de stapel (n>0).

n ROLL ('zet het n-de getal uit de stapel er boven op')
Het totaal aantal stapelelementen verandert niet! (n>1)

Denk er om dat bij de opdrachten PICK en ROLL (dus zonder daarbij n te noemen) voor n het getal gebruikt wordt dat op dat ogenblik op de top van de stapel staat!! Het gebruik van PICK en ROLL zonder n te noemen is daarom beslist af te raden, tenzij je zeker weet wat er op de stapel staat.

Rekenbewerkingen

+ (n1 n2 → som)
Vervang n1 en n2 door hun som.

- (n1 n2 → verschil)
Vervang n1 en n2 door het verschil n1 - n2.

* (n1 n2 → product)
Vervang n1 en n2 door hun product.

/ (n1 n2 → quotient)
Vervang n1 en n2 door het quotient n1/n2 (met weglating van cijfers achter de komma).

MOD (n1 n2 → rest)
Vervang n1 en n2 door de rest (met het teken van n1) bij deling van n1 door n2.

/MOD (n1 n2 → rest quotient)
Vervang n1 en n2 door rest en quotient, berekend als hiervoor.

MAX (n1 n2 → maximum)
Vervang n1 en n2 door het grootste van die twee getallen.

MIN (n1 n2 → minimum)
Vervang n1 en n2 door het kleinste van die twee getallen.

ABS (n → |n|)
Vervang n door de absolute waarde van n.

NEGATE (n → -n)
Draai het teken van n om.

Afdrukken

. (n →)
Druk n (zie voor details paragraaf 7.4) af als een enkele nauwkeurigheid getal, gevolgd door een spatie.

U. (un →)
Als het voorgaande, maar dan zonder teken.

CR (→)
Geef een wagen-terugloop en een nieuwe regel op de schrijfmachine.

Oefeningen

1) Schrijf de volgende expressies in postfix vorm:

```
(1 + 2) * (3 - 4)
10 + 100/9 + 5
2 * (3 * (4 * (5 + 6)))
```

2) Herschrijf de volgende postfix expressies in infix vorm:

```
20 10 + 20 10 - /
1 2 3 4 + + +
20 1 2 * -
```

3) Laat zien hoe de stapel beïnvloed wordt door de volgende bewerkingen, steeds aannemend dat de stapel aanvankelijk leeg was.

```
100 -200 ABS MAX
-10000 0 MIN NEGATE
1 2 SWAP OVER
10 DUP DUP * *
10 20 30 40 3 PICK +
```

4) Hoe zou je som, verschil, product en quotient van twee getallen berekenen en afdrukken, zonder die getallen steeds opnieuw in te tikken? (Aanwijzing: gebruik stapel duplicatie opdrachten).

2 FORTH WOORDEN

Het kan wat ongewoon lijken om al in het begin van het boek te vragen "Wat doet FORTH nu precies wanneer een invoersliert wordt verwerkt?". Maar FORTH is een ongewone taal en het antwoord op deze vraag is niet moeilijk en zal bovendien ons inzicht vergroten en het programmeren in FORTH vereenvoudigen. In dit hoofdstuk wordt een eenvoudig model van een FORTH-systeem beschreven dat een invoersliert verwerkt. Tegelijkertijd wordt terminologie ingevoerd, die vaak in de rest van dit boek wordt gebruikt. Als we weten hoe FORTH werkt, dan kunnen we ook voorzien wat er mis zou kunnen gaan en daarom wordt ook ingegaan op wat je bij fouten moet doen. Tenslotte wordt onze woordenschat uitgebreid met begrippen als variabele, constante en 'array' en hun definitie (in andere programmeertalen gebruikt men vaak het woord declaratie in plaats van definitie; in FORTH Engels spreekt men van 'defining word').

2.1 FORTH in actie

Alle FORTH bewerkingen, die we tot dusver behandeld hebben (en die in de tekst tussen {accoladen} stonden), heten in de terminologie van FORTH *woorden*. Zelfs uit een enkel karakter bestaande bewerkingen als {*} en {.} heten woorden. Ieder woord staat in het FORTH "Woordenboek", zodat wanneer FORTH een invoersliert *interpreteert* (d.i. de betekenis vaststelt met het oog op de toepassing) ieder woord daarvan opgezocht wordt in het Woordenboek. Wat daarin staat voor de woorden {*} en {.} is in figuur 2.1 weergegeven.

```
Woord   Definitie
  *     (n1 n2 → product)   Vermenigvuldig n1 met n2
  .     (n →  )             Druk n af
```

Figuur 2.1 Twee verklaringen in het WOORDENBOEK

Net als in een gewoon woordenboek staan in het FORTH Woordenboek woorden en bij ieder woord een definitie. De definitie geeft aan welke *actie* door dat woord bewerkstelligd wordt bij uitvoering daarvan. Figuur 2.1 laat die actie zien zowel in woorden als met de nauwkeuriger (stapel voor → stapel na) notatie, die in paragraaf 1.6 is ingevoerd. De losse FORTH overzichtskaart geeft op dezelfde manier het FORTH-79 standaard Woordenboek, dat ongeveer 130 woorden bevat.

Neem eens aan dat het woordenboek slechts de woorden {*} en {.} bevat en laten we dan in detail bekijken hoe FORTH het eenvoudige vermenigvuldigingsvoorbeeld interpreteert:

```
23 34 * . 782 ok
```

Na op de return-toets gedrukt te hebben, voert FORTH de volgende handelingen uit:

i) FORTH vindt het eerste woord in de invoersliert, nl. {23} en zoekt dat in het woordenboek op. Omdat het daar niet in staat, neemt FORTH aan dat dit woord een getal is. Dit blijkt een geldig decimaal getal te zijn, zodat het in binaire vorm op de stapel terecht komt.

ii) Evenzo blijkt het tweede woord {34} niet in het woordenboek te staan, maar ook een geldig decimaal getal te zijn, zodat ook dit in binaire vorm op de stapel komt.

iii) Het derde woord in de invoersliert {*} staat wel in het woordenboek, zodat wordt uitgevoerd wat in de definitie staat; het product van de twee bovenste getallen in de stapel vervangt die getallen.

iv) Het vierde woord {.} staat eveneens in het woordenboek en wordt 'uitgevoerd', zodat het getal op de stapel verdwijnt nadat het is afgedrukt.

v) Er is verder niets meer in de invoersliert; FORTH drukt dan de boodschap 'ok' af en wacht op de volgende invoersliert.

We kunnen nu de tweede programmeringsregel in FORTH formuleren:

> *Alle invoer voor FORTH bestaat uit een rij woorden. Ieder woord moet òf in het woordenboek staan, in welk geval het wordt 'uitgevoerd', òf een geldig getal zijn, in welk geval het in binaire vorm op de stapel geplaatst wordt.*

2.2 Foutmelding in FORTH

De voorgaande beschrijving van de werking van FORTH roept de vraag op wat er gebeurt bij een woord dat niet in het woordenboek staat en evenmin een geldig getal is. Welnu als we zo iets intikken als:

```
PQRXYZ  PQRXYZ ?
```

reageert FORTH eenvoudig met de boodschap '*PQRXYZ ?*', die zoals je kunt voorspellen betekent dat {PQRXYZ} niet in het woordenboek staat en ook niet een getal is! De foutmelding '?' lijkt op de foutmelding 'Syntax error' in BASIC, maar bovendien wordt in FORTH het niet-begrepen deel van de invoersliert nog eens afgedrukt, zodat de programmeur kan zien wat FORTH niet begrijpt. Dit is nuttig bij een fout ergens in een lange invoersliert, bijvoorbeeld:

```
1 !2 * 3 4 * +   !2 ?
```

Een andere regel in FORTH is dat ieder woord in de invoersliert door tenminste één spatie gevolgd moet worden. Wordt dit niet nagekomen, dan kunnen duistere fouten daarvan het gevolg zijn, zoals bijvoorbeeld bij het ontbreken van een spatie tussen een getal en een geldig FORTH woord of tussen twee FORTH woorden:

```
23 34* . 34* ?
```

FORTH beschouwt {34*} als een woord, maar kan het niet in het woordenboek vinden of als getal herkennen zodat we de foutmelding '?' ontvangen.

Het ontbreken van een spatie tussen twee getallen is nog erger:

```
2334 * . 0 STACK EMPTY
```

FORTH plaatst na het lezen van het eerste woord het getal 'twee duizend drie honderd en vierendertig' op de stapel. Wanneer FORTH vervolgens de vermenigvuldiging wil uitvoeren, waarvoor twee getallen op de stapel nodig zijn, wordt maar één getal gevonden en de foutmelding ' STACK EMPTY' (Engels voor 'stapel leeg') volgt. Wanneer op de stapel nog getallen stonden van vorige bewerkingen, dan wordt voor de vermenigvuldiging het bovenste daarvan gebruikt en we krijgen een niet-gesignaleerde fout, die desastreus zal zijn in het verdere werk! Het is daarom aan te raden om op geschikte plaatsen de stapel leeg te maken door enkele keren {•} te geven totdat je de boodschap STACK EMPTY krijgt.

Merk ook op dat voor de foutmelding nog een 'valse' nul was afgedrukt. Dit is een gevolg van {•} voor een lege stapel.

De vaak voorkomende foutmelding STACK EMPTY is typisch voor het gebruik van een stapel, zodat er geen equivalent voor is in BASIC. In het algemeen zal deze fout optreden wanneer er minder operanden op de stapel staan dan een FORTH bewerking nodig heeft voor een correcte uitvoering. De stapelnotatie (stapel voor → stapel na), zoals ingevoerd in paragraaf 1.6, leert ons precies hoeveel operanden een bewerking nodig heeft. Bijvoorbeeld:

```
{•}  (n → )
```

heeft één operand nodig,

```
{*}  (n1 n2 → product)
```

heeft er twee nodig, en de volgende drie:

```
{ROT} (n1 n2 n3 → n2 n3 n1)
```

Verderop in dit boek zal een methode beschreven worden, waarmee je de inhoud van de stapel in het oog kunt houden terwijl je een FORTH programma schrijft. Daarmee kun je de kans op STACK EMPTY verkleinen.

2.3 Variabelen in FORTH

In hoofdstuk 1 werd de stapel geïntroduceerd en beschreven hoe in FORTH het rekenwerk met behulp van de stapel wordt uitgevoerd. Voorts zagen we hoe tussenresultaten op de stapel bewaard konden worden voor later gebruik of hoe met manipulatie-opdrachten een (tussen-)resultaat voor meer dan één FORTH bewerking kon worden gebruikt. Een stapel is dus te beschouwen als een nuttig korte-termijn geheugen of als een kladblok voor kleine berekeningen, maar het is kennelijk niet geschikt voor het langer bewaren van getallen. Voor het laatste doel moeten we *variabelen* gebruiken.

De BASIC programmeur is vertrouwd met het gebruik van variabelen omdat daarmee bijna al het rekenwerk wordt gedaan. De BASIC opdracht

```
LET A1=100
```

geeft aan de variabele 'A1' de waarde 100. In FORTH schrijven we dan:

```
100 A1 !
```

maar zou je dit zonder meer intikken, dan verschijnt de foutmelding 'A1 ?'. Dit komt omdat je vooraf de variabele A1 moet definiëren met:

```
VARIABLE A1 ok
```

('ok' wordt natuurlijk door het FORTH systeem geproduceerd). Door deze definitie wordt een geheugenplaats gereserveerd en van het etiket 'A1' voorzien. BASIC kent deze constructie niet omdat in BASIC variabelen 'impliciet' (d.w.z. door ze te noemen) worden gedefinieerd. In FORTH daarentegen moèt een variabele worden gedefinieerd ('gedeclareerd' zou eigenlijk een beter woord zijn) voor zijn gebruik in verdere berekeningen of om er een waarde aan toe te kennen. Voor het definieren van een variabele, geheten <naam>, tikken we:

```
VARIABLE <naam>
```

Opmerking. In sommige FORTH systemen moeten we echter aan de gedefinieerde variabele een 'beginwaarde' n geven met:

```
n VARIABLE <naam>
```

In theorie is er geen beperking in aantal of soort karakters in <naam> (hetgeen door < en > wordt aangegeven). Sommige FORTH systemen vullen het Woordenboek aan met de volledige door de programmeur gekozen <naam>, maar andere systemen bewaren alleen de eerste drie of

vier karakters van <naam> (de documentatie van je systeem vermeldt wel waar je aan toe bent). In het laatste geval moet je er zelf voor zorgen dat namen van variabelen uniek zijn (dus niet Amsterdam en Amstelveen beide invoeren, als jouw systeem alleen de eerste vijf karakters van een naam onthoudt!).

Na de definitie:

```
VARIABLE jaar  ok
```

zal de invoersliert

```
1982 jaar !  ok
```

aan de variabele 'jaar' de waarde 1982 geven. Het getal 1982 zal op de stapel gezet worden. Het woord {jaar} is, volgens het Woordenboek, een FORTH-woord. Het woord {!} zorgt ervoor dat de waarde op de top van de stapel (1982) aan de naam 'jaar' wordt toegekend.

In de FORTH invoersliert

```
jaar @ .  1982 ok
```

zien we het omgekeerde omdat het woord {@} bewerkstelligt dat de waarde van de variabele 'jaar' gedupliceerd wordt op de stapel; door het woord {•} wordt deze waarde dan afgedrukt. De combinatie {@ •} komt zo vaak voor dat daarvoor in FORTH een speciaal woord, {?}, is ingevoerd met precies hetzelfde effect. Met

```
jaar ?  1982 ok
```

kunnen we dus de waarde van de variabele jaar direct afdrukken zonder van de stapel gebruik te maken.

De woorden {!} en {@} (uit te spreken als 'berg op' en 'haal op') kunnen samen gebruikt worden om direct met variabelen te rekenen. Bijvoorbeeld

```
A1 @ 1 + A1 !  ok
```

is equivalent met 'LET A1 = A1 + 1' in BASIC. Dit FORTH voorbeeld is niet zo vreemd als je in aanmerking neemt dat {A1 @} de waarde van A1 op de stapel plaatst en {A1 !} het getal op de stapel opbergt in A1. De invoersliert geeft aanleiding tot drie bewerkingen:

i) Dupliceer de waarde van A1 op de stapel.
ii) Tel 1 op bij de top van de stapel.
iii) Breng het getal op de stapel over naar A1.

Iedere berekening met variabelen kan in FORTH geformuleerd worden. Figuur 2.2 laat bijvoorbeeld zien hoe een BASIC opdracht in FORTH geschreven moet worden, aannemend dat X en Y al gedefinieerd zijn.

```
BASIC:      LET A = 2 * X  +  Y

FORTH:      2 X @  *  Y @ +  A !
```

Figuur 2.2 Rekenen met FORTH variabelen

Het is waar dat rekenen met FORTH variabelen er wat vreemd uitziet, maar aan de andere kant is het ook waar dat in FORTH veel minder met variabelen gewerkt wordt dan in andere talen. Waarom dit zo is, zal later duidelijk worden.

2.4 Variabelen nader bekeken

Waarom we enige tijd besteed hebben aan variabelen, terwijl het hoofdstuk begon met praten over het FORTH Woordenboek, komt omdat de definitie van variabelen een van de vele mogelijke definities is.

Een FORTH Woordenboek moet je vergelijken met een woordenboek waarin met opzet aanvankelijk enkele bladzijden onbedrukt zijn. De programmeur kan daarop nieuwe woorden, samen met hun definitie, schrijven, zodat later bij het opzoeken van een woord zowel de oorspronkelijke als de toegevoegde woorden geraadpleegd worden. Definities zijn dus bijzondere FORTH bewerkingen met het resultaat dat nieuwe woorden op de 'blanco' bladzijden 'bijgeschreven' worden. Zo maakt intikken van:

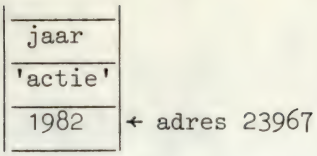
```
VARIABLE jaar ok
```

dat het woord 'jaar' in het Woordenboek wordt bijgeschreven. In paragraaf 2.1 zagen we dat alle FORTH woorden iets bewerkstelligen, wanneer zo'n woord ingetikt wordt en dat geldt ook voor de toegevoegde woorden. Hun actie is namelijk dat het *adres* van de toegevoegde variabele op de stapel wordt geplaatst. Intikken van:

```
jaar . 23967 ok
```

veroorzaakt daarom het afdrukken van een vreemd getal (het is maar verzonnen), dat je waarschijnlijk niet eerder hebt gebruikt. Het is dan ook het werkelijke adres van de geheugenplaats die FORTH gereserveerd heeft voor de variabele {jaar}.

Figuur 2.3 geeft schematisch weer wat aan het Woordenboek toegevoegd is voor het nieuwe woord 'jaar'. Deze toevoeging bestaat uit drie delen. Het eerste is het 'naam' gedeelte, dat dus het woord "jaar" bevat. Het tweede deel geeft de 'actie' van die variabele aan, dus bestaande uit het op de stapel plaatsen van het adres van de momentane waarde van de variabele, welke waarde opgeslagen is in het derde deel. (Preciezer gezegd: in het tweede deel staat het adres van een programma dat deze actie uitvoert.)



Figuur 2.3 Toevoeging aan het Woordenboek voor 'jaar'

Kijken we nu in detail naar de bewerkingen {!} en {@}, dan zien we dat ze equivalent zijn met de BASIC bewerkingen POKE en PEEK. In FORTH zijn ze als volgt gedefinieerd:

```
!    (n adres → )   Berg n op in de geheugenplaats 'adres'
@    (adres → n)    Zet de inhoud van geheugenplaats 'adres'
                         op de stapel
```

(Merk op dat in het eerste geval de stapel twee plaatsen 'inkrimpt' en in het tweede geval even lang blijft omdat een geheugenadres door de inhoud van de genoemde geheugenplaats wordt vervangen.) Zoals bij de meeste FORTH bewerkingen worden door {!} en {@} getallen van 16 bits verplaatst. FORTH heeft ook bewerkingen (zie hoofdstuk 7) voor het verplaatsen van 8 bits, waarmee dan karakters gemanipuleerd kunnen worden.

De FORTH invoersliert

```
1 jaar +!  ok
```

veroorzaakt de volgende reeks bewerkingen:

i) Het eerste woord {1} staat niet in het Woordenboek, moet dus een getal zijn en wordt op de stapel geplaatst.
ii) Het tweede woord {jaar} wordt in het Woordenboek gevonden (omdat het door een voorgaande definitie erin is gezet) en het adres van de variabele 'jaar' komt op de stapel.
iii) Het derde woord {+!} veroorzaakt iets bijzonders: het telt het tweede getal in de stapel op bij de inhoud van de geheugenplaats, waarvan het adres boven op de stapel staat. Dus 'jaar' wordt nu: 1982 + 1 = 1983.

Natuurlijk is het niet nodig om deze reeks bewerkingen te onthouden wanneer je FORTH variabelen gebruikt. Zoals in iedere fatsoenlijke hogere programmeertaal kun je variabelen gebruiken, zonder te weten waar hun waarden worden opgeborgen in het geheugen.

2.5 Constanten in FORTH

'Constanten' vormen een hulpmiddel om vaak gebruikte getallen voor te stellen met behulp van een zinvolle naam in plaats van met een (lange) rij cijfers. Evenals variabelen moeten constanten gedefinieerd worden in FORTH voordat je ze mag gebruiken.

Met bijvoorbeeld

```
1234 CONSTANT telefoonnummer  ok
```

wordt een constante, geheten 'telefoonnummer' en met de waarde 1234, gedefinieerd. Intikken van

```
telefoonnummer .  1234 ok
```

zal plaatsing van 1234 op de stapel veroorzaken alsof het direct ingetikt werd (het is de waarde van de constante en niet het adres ervan), gevolgd door het afdrukken daarvan.

De bewerking {CONSTANT} is in feite weer een voorbeeld van een 'definitie'. In dit voorbeeld wordt de constante {telefoonnummer} weer een nieuw woord in het Woordenboek dat bij verwerking zijn waarde op de stapel plaatst. Die waarde blijft echter altijd de waarde op het moment van definitie; hij mag niet veranderd worden zoals met de waarde van een variabele het geval is (met een berekening). Dit is niet helemaal waar, want als het telefoonnummer moet veranderen, kun je dat doen met

```
5678 CONSTANT telefoonnummer  ok
```

waarmee je een nieuwe constante met dezelfde naam, maar een andere waarde definieert hoewel er al een constante met dezelfde naam is.

Opmerking. Sommige FORTH-systemen geven in dit geval een waarschuwing, waar je je niets van aan hoeft te trekken omdat herdefinitie van een constante een legale bewerking is.

Dat dit geen verwarring geeft, is te danken aan een belangrijke eigenschap van het FORTH Woordenboek:

> *Het opzoeken van een woord in het Woordenboek gebeurt in omgekeerde volgorde van het toevoegen van nieuwe woorden.*

Na de herdefinitie van 'telefoonnummer' hebben we dus wel twee definities van telefoonnummer, maar de laatste definitie zal altijd worden gebruikt.

Hebben we ooit weer de oude definitie nodig, dan kan de meest recente eenvoudigweg 'vergeten' worden met het woord {FORGET}:

```
FORGET telefoonnummer  ok
telefoonnummer .  1234 ok
```

{FORGET} is een nuttige 'huishoudelijke' bewerking, die we o.a. kunnen gebruiken om af en toe het Woordenboek schoon te vegen. Gebruik het echter voorzichtig, want {FORGET <naam>} zal de laatst gedefinieerde versie van <naam> en àlles wat daarna gedefinieerd werd, vergeten.

2.6 Samenvatting en oefeningen

In de hierna volgende beschrijvingen van de stapel heeft 'byte' betrekking op een waarde van 16 bits, waarvan echter alleen de laatste 8 bits in acht worden genomen door een bewerking; de eerste 8 bits worden meestal nul gemaakt. De afkorting 'adr' heeft betrekking op een getal van 16 bits dat het adres van een byte in het geheugen is. De zo geadresseerde byte kan het eerste onderdeel zijn van een groter geheel (als een variabele van 16 bits).

Met <naam> wordt het volgende woord, voorafgegaan en gevolgd door spaties, in de invoersliert bedoeld; dit woord mag bestaan uit willekeurige karakters - behalve de spatie - van de standaard ASCII verzameling van karakters, maar soort en aantal van karakters is voor een bepaald systeem soms beperkt voor nieuwe woorden in het Woordenboek. (Voor een beschrijving van ASCII raadplege men de verklarende woordenlijst achter in het boek.)

Geheugenbewerkingen

@ (adr → n)
Zet het getal in de geheugenplaats 'adr' op de stapel.

! (n adr →)
Zet het getal onder de top van de stapel in de geheugenplaats 'adr'.

C@ (adr → byte)
Zet de byte in de geheugenplaats 'adr' op de stapel.

C! (byte adr →)
Zet de byte onder de top van de stapel in de geheugenplaats 'adr'.

? (adr →)
Druk het getal in de geheugenplaats 'adr' af

+! (n adr →)
Tel n op bij het getal van 16 bits in de geheugenplaats 'adr'.

Definitiebewerkingen

VARIABLE (→)
Door VARIABLE <naam> wordt het woord <naam> toegevoegd aan het Woordenboek op de eerder beschreven wijze (zie hoofdstuk 9 voor meer details). Wanneer <naam> later wordt uitgevoerd, zal het zijn geheugenadres op de stapel plaatsen:
<naam> (→ adr)

CONSTANT (n →)
Door n CONSTANT <naam> wordt de constante <naam> toegevoegd aan het Woordenboek, samen met de waarde n. Wanneer <naam> later wordt uitgevoerd, komt n op de stapel terecht:
<naam> (→ n)

Woordenboekbeheer

FORGET (→)

Door FORGET <naam> wordt de allerlaatste toevoeging aan het Woordenboek van <naam> weer verwijderd evenals alle sindsdien toegevoegde woorden. Er wordt een foutmelding geproduceerd als <naam> niet te vinden is.

Oefeningen

1) Definieer de volgende constanten:

naam	*waarde*
tien	10
fred	4 * tien + 1

2) Definieer de volgende variabelen en geef ze de opgegeven waarden:

naam	*beginwaarde*
XYZ	-100
A-	XYZ - fred

3) Schrijf een met de volgende BASIC opdracht equivalente FORTH opdracht, aannemend dat de variabele X al gedefinieerd is:

```
LET  X = 1 + X + X * X
```

Kun je je oplossing nog verbeteren door gebruik te maken van DUPlicatie of zelfs van {+!} ?

4) Schrijf een FORTH expressie voor het berekenen van

$ax^2 + bx + c$

waarin x al gedefinieerd is als een FORTH variabele en de constanten a, b en c gedefinieerd zijn als FORTH constanten.

5) Laat zien hoe {CONSTANT} gebruikt kan worden om een willekeurige geheugenplaats een naam te geven en vervolgens die naam als een variabele te gebruiken. Let op: probeer dit niet echt op je machine omdat je daarmee het systeem voor goed zou kunnen bederven!!

3 DE 'COLON' -DEFINITIE

Tot nu toe hebben we twee definitie-opdrachten, {VARIABLE} en {CONSTANT} in detail bekeken. Beide voegen woorden aan het Woordenboek toe en voeren bepaalde acties uit wanneer die woorden als opdracht worden gebruikt. Dit doet ook de 'Colon'-definitie {:}, die echter een veel algemener effect heeft, nl. het zelf kunnen definiëren van de actie van het nieuwe woord.

3.1 'Colon' berekeningen

Neem eens aan dat we met FORTH wat percentages willen berekenen. Dit zou kunnen met bijvoorbeeld:

```
150 12 * 100 / . 18 ok
```

om 12% van 150 uit te rekenen. Voor het berekenen van veel percentages zou echter het gebruik van een 'percentage operator' eenvoudiger zijn (ongeveer als het benutten van een 'procent-toets' op zakrekenmachines). Zo'n operator is met een eenvoudige colon-definitie als volgt te definiëren:

```
: % * 100 / ; ok
```

zodat we voortaan in plaats van {* 100 /} slechts {%} hoeven te tikken:

```
150 12 % . 18 ok
```

Dit is veel netter, vergt minder tikken (en vermindert daarmee de kans op fouten) en is beter leesbaar.

Deze definitie van {%} is een voorbeeld van een colon-definitie, die een woord aan het Woordenboek toevoegt dat bij uitvoering hetzelfde effect heeft alsof {* 100 /} in zijn geheel ingetikt was. Figuur 3.1 laat zien wat de structuur van de colon-definitie is.

```
:      %   * 100 /     ;
↓      ↓   └─────┘     ↓
begin  naam  lichaam   einde
```

Figuur 3.1 De Colon-definitie van {%}

De colon (dubbele punt) {:} en de kommapunt {;} geven begin en einde van de definitie aan; zij mogen niet weggelaten worden! Het <naam>-deel is het eerste woord na de colon; het komt als zodanig in het Woordenboek.

Zoals in iedere invoersliert moet tussen de colon en de naam tenminste één spatie staan, evenals tussen de naam en het eerste woord van het 'lichaam'. Het lichaam van de definitie kan iedere reeks van FORTH woorden en getallen zijn; het wordt in vertaalde vorm als uit te voeren actie in het Woordenboek opgenomen. Die actie vindt plaats als de corresponderende naam in een invoersliert tegengekomen wordt.

De naam in een colon-definitie mag, evenals bij variabelen en constanten, uit een willekeurig aantal karakters bestaan. Alle soorten karakters mogen gebruikt worden. In de praktijk vereisen veel FORTH-systemen dat slechts de eerste drie of vier karakters, van twee verschillende namen, verschillend (uniek) zijn.

In het {%} voorbeeld bestaat de actie enkel uit het vermenigvuldigen van twee getallen op de top van de stapel en het door 100 delen van het product. We kunnen het nieuwe woord {%} dus net zo beschrijven als iedere rekenbewerking, nl. met behulp van onze stapel-notatie (stapel voor → stapel na):

```
%          (n1 n2 → procent)          procent = n1 * n2 / 100
```

Het is inderdaad een goed idee om alle nieuwe woorden in het Woordenboek op deze manier te documenteren, zodat je altijd precies weet wat je uitgebreide woordenschat bevat en hoe je het woord kunt gebruiken. Dit is vooral nuttig waneer je die nieuwe woorden opnieuw gaat gebruiken in latere definities.

3.2 Nog meer over percentages

Is {%} eenmaal gedefinieerd, dan kun je het, net als alle FORTH opdrachten, gebruiken in ingewikkelder expressies als:

```
500 15 % 2 % . 1 ok
```

(waarmee 2% van 15% van 500 wordt berekend) of opnemen in het lichaam van een andere colon-definitie, bijvoorbeeld:

```
VARIABLE rekening ok
: investeer
     rekening @  ( rekening op de stapel )
     12 %        ( bereken de rente )
     rekening +! ( tel die rente op bij rekening )
; ok
```

Zetten we dan $ 200 op onze rekening met:

```
200 rekening ! ok
```

dan zal met samengestelde interest die rekening na drie jaar groot zijn:

```
investeer investeer investeer ok
rekening @ . 280 ok
```

Dit voorbeeld laat een aantal nieuwe mogelijkheden zien:

i) Een colon-definitie mag uit meer dan één invoersliert bestaan; zelfs al tikken we een 'return' in aan het eind van iedere sliert, toch zal FORTH pas na de afsluitende kommapunt de definitie als beëindigd opvatten en 'ok' afdrukken. Naar eventuele extra spaties wordt niet gekeken, zodat die naar believen toegevoegd kunnen worden om de leesbaarheid van de tekst te vergroten.

ii) Verklarend commentaar mag, tussen ronde haakjes geplaatst, aan het eind van iedere regel toegevoegd worden. Onthoud voortaan echter dat 'haakje open' {(} een 'woord' is in FORTH, zodat net als bij alle FORTH woorden aan beide kanten van dat haakje tenminste één spatie moet staan. Het 'haakje sluiten' is geen FORTH woord, maar een 'afsluiter' om het eind van het commentaar aan te geven.

iii) Eerder gedefinieerde variabelen en constanten mogen in het lichaam van een colon-definitie opgenomen worden. Het zijn immers gewoon woorden in het Woordenboek.

Als we de definitie van {investeer} nader bekijken, zien we dat dit woord niet een nieuwe 'rekenbewerking' is zoals {%}. Het is in feite een volledig, zij het erg eenvoudig, programma. Je kunt het uitvoeren door in te tikken:

```
investeer ok
```

De stapel wordt tijdens de uitvoering van {investeer} gebruikt, maar overigens niet veranderd! Niettemin kunnen we {investeer} en {rekening} op de aanbevolen manier documenteren:

```
rekening      ( → adres)    Variabele voor 'investeer'
investeer     ( →  )        Tel 12% rente op bij rekening
```

Merk op dat deze nieuwe woorden werden gedocumenteerd in de volgorde van hun definitie.

3.3 Colon definitie of programma?

Figuur 3.2 laat een heel eenvoudig BASIC programma en een equivalente FORTH definitie zien.

```
BASIC                                FORTH

                                     : kwadraat
10 INPUT X                                    ." = "
20 PRINT "kwadraat =  "; X*X                  DUP * .
                                     ; ok
RUN
?4
kwadraat = 16                        4 kwadraat = 16 ok
```

Figuur 3.2 BASIC en FORTH

Regel 10 van het BASIC programma vraagt de gebruiker een getal in te tikken, dat toegekend wordt aan de variabele 'X'. Regel 20 drukt de tekst "kwadraat = " en vervolgens het product X*X af.

Het FORTH equivalent is eenvoudiger omdat in plaats van de gebruiker een te kwadrateren getal te vragen, FORTH het getal op de top van de stapel gebruikt voor de kwadratering. Met voorbedachte rade heeft de colon-definitie de naam {kwadraat} gekregen, zodat we voor het verwerken van het programma slechts een getal hoeven in te tikken, gevolgd door "kwadraat" en het indrukken van de return-toets. De actie van {kwadraat} bestaat uit het eerst afdrukken van " = ", dan de kwadratering van het getal op de top van de stapel en tenslotte het afdrukken van het resultaat. Dit geeft een verrassend fraaie en leesbare manier om het programma uit te voeren; bijvoorbeeld:

```
4 kwadraat = 16 ok
5 kwadraat = 25 ok
6 kwadraat = 36 ok
```

Merk op dat wanneer je vergeten had het te kwadrateren getal in te tikken, FORTH gereageerd zou hebben met een foutmelding:

```
kwadraat = 0 STACK EMPTY
```

Dit omdat {kwadraat} een getal nodig heeft, zoals te zien met

```
kwadraat        (n → )        Druk n² af.
```

Twee verdere kenmerken van {kwadraat} verdienen de aandacht:

i) {kwadraat} heeft in tegenstelling tot het BASIC equivalent geen variabele nodig. Het is karakteristiek voor FORTH programma's dat variabelen niet zo vaak gebruikt worden omdat voor tussenwaarden bij voorkeur de stapel wordt gebruikt.

   Ervaren programmeurs zullen het nut van stapelgebruik mogelijk betwijfelen en zich afvragen of FORTH een equivalent heeft voor de INPUT opdracht van BASIC. Zoals we in hoofdstuk 7 zullen zien is het in FORTH mogelijk om invoer van de gebruiker te vragen, maar in de meeste toepassingen is stapelgebruik voor het doorgeven van gegevens voor een programma te verkiezen. Het is zeker gemakkelijker en wat is fraaier dan alleen '4 kwadraat' te hoeven intikken.

ii) Het gebruik van {."} in FORTH is het equivalent van PRINT "...." in BASIC; het zal alle tekst na {."} tot het volgende aanhalingsteken " afdrukken. In de meeste FORTH-systemen mag je dit ook buiten de colon-definitie gebruiken, bijvoorbeeld:

```
." hallo vriend "          hallo vriend ok
```

Gebruik je dit binnen een colon-definitie, dan wordt de desbetreffende tekst vertaald en pas afgedrukt als je de naam in de colon-definitie laat uitvoeren. Bijvoorbeeld:

```
: GROET ." hallo vriend " ; ok
GROET hallo vriend  ok
```

Let er op dat {."} voorafgegaan en gevolgd moet worden door tenminste één spatie. De afsluitende " hoeft niet voorafgegaan te worden door een spatie. Eventuele spaties vormen hier een deel van de af te drukken tekst.

We kunnen nu de vraag, vervat in de titel van deze paragraaf, beantwoorden door op te merken dat programmeren in FORTH gebeurt door één of meer colon-definities op te stellen. M.a.w. een colon-definitie is een programma.

Vergelijking van een BASIC programma en een colon-definitie in FORTH illustreert dit principe, maar de analogie moet je niet te ver vervolgen. De BASIC programmeur ontwikkelt, corrigeert en verfraait één programma, terwijl de FORTH programmeur hetzelfde doel bereikt door het opstellen van een reeks colon-definities, die ieder afzonderlijk getest worden en dan vertaald en toegevoegd aan het FORTH Woordenboek. Om een aantal verschillende BASIC programma's uit te voeren, moet je die afzonderlijk laden en uitvoeren; vertaalde FORTH programma's zijn daarentegen zo compact dat ze tegelijk in het Woordenboek kunnen voorkomen en een willekeurig programma daaruit kan worden uitgevoerd door alleen maar de naam ervan in te tikken.

3.4 Interpreteren of vertalen?

Nu is het moment gekomen om dieper in te gaan op de begrippen 'interpreteren' en 'vertalen', die we al op een aantal plaatsen in dit boek zijn tegengekomen. Laten we daartoe de volgende aftrekking bekijken:

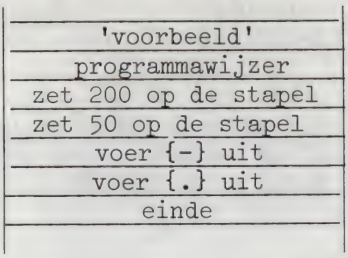
```
200 50 - . 150 ok
```

Zodra we de invoersliert ingetikt hebben en de return-toets hebben ingedrukt, zal FORTH de invoersliert interpreteren op de reeds beschreven manier. Ieder woord in de sliert wordt opgezocht in het Woordenboek; wordt het daar gevonden dan wordt de corresponderende actie uitgevoerd, staat het niet in het Woordenboek dan wordt het woord behandelt als ware het een getal en komt het dus op de stapel.

Iedere FORTH invoersliert, die zo ingetikt en uitgevoerd kan worden, kan ook opgenomen worden in een colon-definitie door er eenvoudigweg {: <naam>} vóór en {;} àchter te zetten. De dubbele punt heeft tot gevolg dat FORTH in plaats van te interpreteren overgaat tot het vertalen van de daarop volgende tekst; de kommapunt doet net het omgekeerde. Behandelen we de eerder gegeven aftrekking op die manier, dan krijgen we de volgende colon-definitie:

```
: voorbeeld 200 50 - . ; ok
```

De dubbele punt veroorzaakt het opnemen van het woord 'voorbeeld' in het Woordenboek en dwingt FORTH tot het vertalen van alles wat nu volgt tot de kommapunt. Het resultaat is een compacte reeks (machine-) opdrachten die bij het woord 'voorbeeld' worden geplaatst, zoals schematisch aangegeven in figuur 3.3:



Figuur 3.3 Een toevoeging aan het Woordenboek

De toevoeging aan het Woordenboek begint met de naam van het nieuwe woord (hier dus 'voorbeeld'). Daarna volgen vier opdrachten, ieder corresponderend met een woord in het lichaam van de colon-definitie. Om deze instructies uit te voeren, tikken we slechts in:

```
voorbeeld 150 ok
```

om hetzelfde resultaat te krijgen als met de geïnterpreteerde invoersliert.

In werkelijkheid nemen de vertaalde opdrachten niet zo veel ruimte in als uit figuur 3.3 zou lijken. Iedere opdracht bestaat slechts uit het 'adres' van het corresponderende woord in het Woordenboek plus nog wat informatie die pas in hoofdstuk 9 nader wordt besproken. De programmawijzer wijst naar een snel uitvoeringsprogramma dat de woorden in de definitie uitvoert met behulp van die adressen. Regel is dus dat:

> *Een FORTH invoersliert meteen wordt geïnterpreteerd en uitgevoerd, maar als dezelfde invoer wordt opgenomen in een colon-definitie, dan wordt die vertaald en desgewenst later uitgevoerd door de naam van de colon-definitie in te tikken.*

3.5 Het maken van tabellen en arrays

Een array bestaat uit een reeks gelijksoortige grootheden. Een gemeenschappelijke eis in veel programma's is dat ruimte gemaakt moet worden voor arrays. Mogelijkheden daarvoor worden in de meeste programmeertalen geboden. In BASIC worden de 'afmetingen' (d.i. het aantal rijen en/of kolommen) gegeven met behulp van een 'DIM'-opdracht.

In FORTH is geen equivalent voor 'DIM' in de standaard woordenschat, maar toch is voorzien in alle benodigde bewerkingen om 'arrays te bouwen' wanneer ze nodig zijn. (Denk er aan dat FORTH een uitbreidbare taal is, zodat niet al vooraf voorzien hoeft te worden in alle mogelijkheden die je misschien ooit nodig zult hebben.)

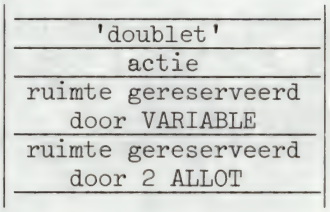
De eenvoudigste manier om voor een array ruimte te reserveren in het Woordenboek is het gebruiken van het woord {ALLOT} ("toewijzen" zou de Nederlandse vertaling van dit woord zijn).

```
ALLOT     (n →  )    Wijs n extra bytes toe aan de aller-
                     laatste toevoeging aan het Woordenboek.
```

{ALLOT} kan samen met {VARIABLE} worden gebruikt. De eerste twee woorden uit:

```
VARIABLE doublet 2 ALLOT ok
```

zullen tot gevolg hebben dat een nieuwe variabele, genaamd 'doublet', met ruimte voor één enkele waarde gedefinieerd wordt. Het deel {2 ALLOT} reserveert daar echter twee extra bytes bij, zodat de variabele 'doublet' in totaal ruimte heeft voor twee waarden. (Een enkele waarde vergt immers 2 bytes geheugenruimte.) In figuur 3.4 is dit nog eens schematisch aangegeven.



Figuur 3.4 Toevoeging aan het Woordenboek voor {doublet}

We hebben nu in feite een array met twee elementen. Het woord {doublet} zal het adres van het eerste getal in het array opleveren en door hier 2 bij op te tellen, krijgen we het adres van het tweede getal. Bijvoorbeeld:

```
100 doublet ! ok
200 doublet 2+ ! ok
```

zal het array initialiseren (een deftig woord voor beginwaarden geven) met de waarden 100 en 200;

```
1 doublet 2+ +! ok
doublet 2+ ? 201 ok
```

vermeerdert het tweede array-element met 1 en drukt het af.

Een alternatieve, en in veel opzichten elegantere, manier om array-ruimte te reserveren, maakt gebruik van de definitie met {CREATE}, die wel een naam in het Woordenboek plaatst maar verder daarin geen ruimte reserveert. Het laatste kan wel in combinatie met {ALLOT} gebeuren. Met bijvoorbeeld:

```
CREATE array 40 ALLOT ok
```

wordt een array met plaats voor 20 getallen gedefinieerd.

Opmerking. Pas op! Op sommige FORTH-systemen kan {CREATE} niet worden gebruikt en moet je de eerder beschreven methode gebruiken, dus:

```
VARIABLE array 38 ALLOT ok
```

Raadpleeg daarom de documentatie van je eigen systeem!

Willen we niet alleen ruimte reserveren, maar in die ruimte meteen getallen zetten, dan moeten we in plaats van {ALLOT} de opdracht {,} gebruiken. Deze opdracht brengt het getal op de top van de stapel over naar de volgende vrije plaats (2 bytes) in het Woordenboek. Dit is erg handig voor het maken van een tabel met constanten, zoals in

```
CREATE tabel -10 , -5 , 1 , 4 , 9 , ok
```

(denk om de spaties aan weerskanten van de komma!) waarmee een array met de vijf elementen -10, -5, 1, 4 en 9 gemaakt is. Om hier een willekeurig element uit te halen, kunnen we natuurlijk als bij 'doublet' het benodigde cijfer optellen bij het door {tabel} te leveren adres, maar een slimme colon-definitie geeft een elegantere methode:

```
: tabel@ 1- 2 * tabel + @ ; ok
```

Om dan een bepaald element uit het array te halen, tikken we diens volgorde-nummer in gevolgd door {tabel@}; bijvoorbeeld:

```
5 tabel@ . 9 ok      (drukt het vijfde element af)
1 tabel@ ok          (zet het eerste element op de stapel)
2 tabel@ ok          (en ook het tweede)
+ . -15 ok           (en druk de som van die twee af)
```

Let op het gebruik van de opdracht {1-} in de definitie van {tabel@}, en van de opdracht {2+} in de eerdere voorbeelden met {doublet}. Voor het gemak van de programmeur zijn in FORTH vier vaak gebruikte optellingen en aftrekkingen gedefinieerd, nl. {1+}, {1-}, {2+} en {2-}. Ze hebben precies hetzelfde effect als de langere {1 +}, {1 -}, {2 +} en {2 -}, maar hebben het voordeel dat ze vertaald kunnen worden naar snel uit te voeren instructies in de machinetaal.

3.6 Uitbreiding van de stapel-notatie

In het vorige hoofdstuk is een methode beloofd waarmee je de inhoud van de stapel in het oog kunt houden tijdens het schrijven van een programma. Deze methode komt ook goed van pas voor het uitleggen van de werking van de voorgaande opdracht {tabel@}. De methode komt neer op het onder elkaar opschrijven van ieder woord in het lichaam van de colon-definitie. Schrijf vervolgens achter ieder woord wat er volgens de losse overzichtskaart met de stapel gebeurt en houdt daarbij rekening met het feit dat 'stapel na' van een woord de 'stapel voor' van het volgende woord wordt. Figuur 3.5 toont deze methode voor de beschrijving van de werking van {tabel@}.

woord	*effect op stapel*	*toelichting*
1-	(n → n-1)	1 af van index
2	(n-1 → n-1 2)	2 op stapel
*	(n-1 2 → adresverschil)	vermenigvuldig
tabel	(adresverschil → adresverschil adres)	haal tabel.adres
+	(adresverschil adres → element.adres)	tel op
@	(element.adres → m)	haal element

Figuur 3.5 De werking van {tabel@}

Een gevolg van deze methode is dat 'stapel voor' van het eerste woord en 'stapel na' van het laatste woord het uiteindelijke effect op de stapel is van {tabel@}, zodat deze opdracht formeel beschreven is met:

```
tabel@   (n1 → n2)   Haal uit het array 'tabel' het element
                     geïndexeerd met n1
```

Deze stapel-notatie is van onschatbare waarde bij het ontwikkelen van ingewikkelde FORTH programma's, ook omdat je de losse overzichtskaart niet meer zo vaak hoeft te raadplegen wanneer je je eigen 'woorden' op die manier netjes documenteert. De boven geschetste methode lijkt omslachtig, maar is snel aangeleerd met wat oefening.

Een andere nuttige methode voor het opsporen van fouten in FORTH programma's maakt gebruik van de opdracht {DEPTH}, samen met {.}. Tijdens het ontwikkelen van programma's kunnen we daarmee op zelf te kiezen kritische punten in het programma het aantal getallen op de stapel afdrukken (en vergelijken met wat wij verwachten). We definieren daartoe een hulpopdracht met

```
: .s  CR DEPTH .  ; ok
```

die het aantal getallen op de stapel afdrukt zonder de stapel te wijzigen.

Deze hulpopdracht kunnen we dan gebruiken in de definitie:

```
: tabel@ 1- 2 * .s tabel + .s @ ; ok
```

Door eerst de stapel leeg te maken en dan deze laatste opdracht uit te voeren, krijgen we:

```
. . . 0 STACK EMPTY
4 tabel@
1
1 ok
```

zoals te verwachten. In de definitieve versie van {tabel@} laten we natuurlijk .s weg omdat die alleen tijdens het testen van {tabel@} nodig is om te controleren hoeveel getallen in de stapel zaten.

3.7 Samenvatting en oefeningen

In dit hoofdstuk zijn de volgende nieuwe opdrachten (woorden) ingevoerd:

Stapel manipulatie

DEPTH (→ n)
Zet het oorspronkelijk aantal getallen in de stapel er boven op.

Rekenbewerkingen

1+ (n → n+1)
Telt 1 op bij het getal boven op de stapel.

1- (n → n-1)
Trekt 1 af van het getal boven op de stapel.

2+ (n → n+2)
Telt 2 op bij het getal boven op de stapel.

2- (n → n-2)
Trekt 2 af van het getal boven op de stapel.

Definitiebewerkingen

: (→)
Met behulp van een colon-definitie:

```
: <naam> . . . . ;
```

wordt <naam> in het Woordenboek gezet en de opdrachten na <naam> worden vertaald naar een programma in de machinetaal dat achter <naam> ook in het Woordenboek komt. Dit programma wordt uitgevoerd wanneer <naam> later wordt genoemd.

; (→)
'Afsluiter' van een colon-definitie; wat er na komt wordt weer geïnterpreteerd.

CREATE (→)

Met behulp van CREATE <naam> wordt <naam> in het Woordenboek gezet en daarachter ruimte gereserveerd. Wordt later <naam> uitgevoerd, dan komt het adres van die ruimte op de stapel:

<naam> (→ adres)

Woordenboekbeheer

ALLOT (n →)

Reserveer n bytes achter de laatste toevoeging aan het Woordenboek.

, (n →)

Gebruik 2 bytes van de laatst gereserveerde ruimte in het Woordenboek om het getal op de top van de stapel ernaar over te brengen.

Afdrukken en diversen

." (→)

Met behulp van ." <tekst> " wordt <tekst> afgedrukt (niet de afsluitende "). Deze 'tekst' mag ten hoogste 127 karakters bevatten. Wanneer deze constructie in een colon-definitie staat, gebeurt het afdrukken pas tijdens de uitvoering van het gedefinieerde woord.

((→)

De constructie (<tekst>) (denk om de spatie achter en voor de 'openingshaak' en achter de 'sluithaak'!!) wordt door FORTH behandeld alsof die helemaal niet bestond.

Oefeningen

1) Schrijf een colon-definitie voor het snel verdrievoudigen van het getal op de top van de stapel.

2) Schrijf een colon-definitie, genaamd 'par', die na 2 blanco regels voor aan een nieuwe regel zal afdrukken "Paragraaf", gevolgd door het getal op de top van de stapel.

3) Maak een array voor vier waarden, die aanvankelijk -10, 1, 10 en 1000 zijn. Definieer ook een opdracht om het adres van de i-de waarde op de stapel te zetten, waarbij i een van de getallen 0, 1, 2 of 3 kan zijn.

4) Definieer een opdracht die ieder van de vier waarden in het array van de vorige opgave zal verdubbelen.

5) Gebruik de methode beschreven in paragraaf 3.6 om het effect van de volgende colon-definitie te verklaren en de bijbehorende stapelnotatie te geven (aanwijzing: de stapel moet aanvankelijk twee of meer getallen bevatten).

```
: voorbeeld DUP * SWAP DUP * + ;
```

4 FORTH STRUCTUREN 1: IF

Voor de BASIC opdracht 'GOTO' bestaat geen equivalent in FORTH, maar omdat FORTH een gestructureerde taal is, is er ook geen behoefte aan deze opdracht. Dit is geenszins overdreven. Programma's moeten goed leesbaar zijn, moeten zichzelf goed documenteren en moeten vooral goed gestructureerd zijn. Wat betekent het laatste precies? Drie kenmerken zijn daarvoor maatgevend:

i) Het is mogelijk een groep opdrachten te formuleren, waarin alle opdrachten na elkaar worden uitgevoerd.
ii) Afhankelijk van het wel of niet voldoen aan een voorwaarde wordt of de ene of de andere groep opdrachten uitgevoerd.
iii) Het is mogelijk een groep opdrachten net zo vaak te laten uitvoeren totdat aan een bepaalde voorwaarde niet meer voldaan wordt of zolang aan een bepaalde voorwaarde voldaan wordt.

FORTH heeft deze drie kenmerken, waarvan het eerste al in alle tot dusver gegeven voorbeelden gebleken is. Aan de tweede voorwaarde wordt voldaan dank zij de te behandelen IF . . ELSE . . THEN structuur, aan de derde door de in het volgende hoofdstuk te bespreken 'lus-structuren': de DO-lus, de UNTIL-lus en de WHILE-lus. In dit hoofdstuk komt de IF-structuur aan de orde en de daarbij behorende vergelijkings- en logische bewerkingen.

4.1 True of false?

Een opdracht die het teken van een getal zal onderzoeken en moet bevestigen of dat getal negatief is of niet, zou als volgt gedefinieerd kunnen worden:

```
: Negatief? 0 < IF ." ja " THEN ; ok
```

We kunnen het dan gebruiken in bijvoorbeeld:

```
-20 Negatief? ja ok
20 Negatief? ok
```

Fraaier zou de volgende verfijning van deze opdracht zijn:

```
: Negatief? 0 < IF ." ja " ELSE ." nee " THEN ; ok
```

die een 'symmetrischer' antwoord zou geven:

```
-1 Negatief? ja ok
1 Negatief? nee ok
```

Laten we de laatste definitie van {Negatief?} eens bekijken. Bij uitvoering van deze opdracht wordt eerst 0 op de stapel gezet. Het woord {<} vergelijkt vervolgens de twee bovenste getallen op de stapel en vervangt ze door de waarde *true* wanneer het bovenste getal groter is dan het getal daaronder, en door de waarde *false* wanneer dat niet het geval is. Het volgende woord {IF} haalt die bovenste waarde weer van de stapel weg en zorgt er voor dat de direct daarna volgende opdrachten worden uitgevoerd (tot een {ELSE}) wanneer de weggehaalde waarde 'true' was; en anders de opdrachten achter {ELSE} tot een {THEN}. In beide gevallen wordt daarna de achter {THEN} staande opdracht uitgevoerd; {THEN} werkt als 'afsluiter' van de IF-constructie en mag nooit weggelaten worden. Wel mag je het {ELSE}-deel (met de opdrachten daarachter), als dat zo uitkomt, weglaten zoals we in het eerste voorbeeld zagen.

Het woord {<} behoort tot de klasse van 'vergelijkings'-woorden, die als regel direct voor een {IF} staan. De formele definitie van {<} is:

```
<          (n1 n2 → b)   b wordt 'true' als n1 < n2,
                         anders wordt b 'false'
```

Hierin zijn n1 en n2 willekeurige enkele-nauwkeurigheid-getallen, terwijl b (in FORTH ook wel 'flag' genoemd) ook een getal is dat echter een van de twee logische waarden 'false' of 'true' voorstelt (in onze taal zou je over 'waar' en 'onwaar' kunnen spreken, maar dit is niet gebruikelijk omdat deze woorden ook in onze dagelijkse taal voorkomen). De samenhang tussen logische en numerieke waarden is als volgt:

logische waarde	*numerieke waarde*
false	0
true	1

zoals ook uit de volgende voorbeelden blijkt (let ook op de postfix vorm!):

```
-2 4 < . 1 ok
20 10 < . 0 ok
```

4.2 Definitie van de IF-structuur

```
conditionele woorden
          IF
                'true-opdrachten'
          ELSE
                'false-opdrachten'
          THEN
```

Figuur 4.1 De volledige IF-structuur

De bovenstaande figuur geeft een beeld van de volledige IF-structuur. Hierin worden de 'true-opdrachten' uitgevoerd wanneer de van de stapel afgehaalde waarde *true* (of niet 0) is. Is die waarde *false* (of 0), dan worden de 'false-opdrachten' uitgevoerd.

Voor de werking van de IF-opdracht moet een logische waarde op de stapel staan, zodat deze opdracht meestal vooraf wordt gegaan door een 'vergelijkings'-woord als {<}. Noodzakelijk is dit niet omdat in FORTH de IF-opdracht ieder van 0 verschillend getal als 'true' opvat en alleen het getal 0 als 'false'.

Opmerking. Een opdracht om te onderzoeken of een getal op de stapel niet-nul is, kan daarom zijn:

```
: niet-nul? DUP IF ." ja" ELSE ." nee" THEN ; ok
34 niet-nul? ja ok
```

De 'true-opdrachten' en 'false-opdrachten' mogen bestaan uit een willekeurig aantal FORTH opdrachten met inbegrip van {IF . . ELSE . . THEN} opdrachten. Op die manier kunnen IF-structuren in principe dus willekeurig 'diep' 'genest' worden (denk ter verontschuldiging van dit woord aan een 'nest' schalen).

Zoals aangegeven in het eerste voorbeeld van dit hoofdstuk, is het 'ELSE'-deel met de daarop volgende 'false-opdrachten' niet verplicht, zodat je het weg kan laten als het niet nodig is.

Belangrijk is te weten is dat *de IF-structuur en in feite alle structuren, die in dit en het volgende hoofdstuk worden besproken, uitsluitend in een colon-definitie gebruikt mogen worden!* Voor insiders zij vermeld dat dit komt omdat deze structuren 'voorwaartse sprongen' impliceren; deze zijn pas volledig bekend na vertaling van de colon-definitie.

Figuur 4.2 laat de samenhang zien tussen een IF-opdracht in BASIC en die in FORTH en in het bijzonder de herrangschikking die karakteristiek is voor FORTH.

```
BASIC:    IF A=2 THEN PRINT "A=2"
FORTH:    A @ 2 = IF ." A=2" THEN
```

Figuur 4.2 BASIC IF → FORTH IF

De FORTH opdracht in figuur 4.2 moet een deel zijn van een colon-definitie en de variabele A moet al eerder gedefinieerd zijn. Bovendien zien we hierin een ander vergelijkingswoord {=}, dat de twee bovenste getallen op de stapel vervangt door een logische waarde en wel 'true' alleen wanneer de te vergelijken getallen inderdaad gelijk zijn.

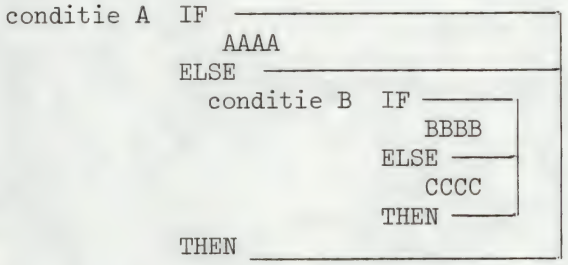
4.3 Geneste IF-structuren

In figuur 4.3 zien we een colon-definitie met twee geneste IF-structuren. Het gedefinieerde woord 'uitslag' zal bij uitvoering een van de drie mogelijkheden 'afgewezen', 'geslaagd' of 'met lof' afdrukken, afhankelijk van het cijfer op de top van de stapel.

```
: uitslag DUP 6 < IF
                    ." afgewezen"       ( minder dan 6)
                   DROP
                ELSE
                     9 < IF
                             ." geslaagd"  ( 6-8)
                         ELSE
                              ." met lof" (9 of 10)
                         THEN
                THEN ; ok
```

Figuur 4.3 De definitie van {uitslag}

De tweede IF-structuur is geheel opgenomen in het ELSE-deel van de buitenste IF-structuur zoals nog eens weergegeven in figuur 4.4:



Figuur 4.4 Geneste IF-structuren

Wanneer zoals in figuur 4.4 met elkaar corresponderende conditionele woorden met elkaar verbonden worden, dan mogen de verbindingslijnen elkaar niet kruisen! Is dit wel het geval, dan is niet alleen het programma zinloos, maar is het ook geen correct FORTH. Het is gebruikelijk om bij geneste structuren in te springen om de leesbaarheid voor ons te verhogen (computers hebben geen moeite met wel of niet inspringen!). BASIC programmeurs zijn dit niet gewend.

Wanneer in figuur 4.4 de conditie A 'true' oplevert, worden de woorden AAAA uitgevoerd. Gaf conditie A daarentegen 'false', dan wordt conditie B bekeken, resulterend in uitvoering van hetzij BBBB, hetzij CCCC. Weer kijkend naar figuur 4.3 zal het volgende duidelijk zijn:

```
3 uitslag afgewezen ok
8 uitslag geslaagd ok
9 uitslag met lof ok
```

Let op de {DUP}-opdracht voor conditie A, die noodzakelijk is voor conditie B omdat uitvoering van conditie A het oorspronkelijke cijfer van de stapel verwijderd heeft. De opdracht {DROP} zorgt er voor dat het oorspronkelijke cijfer ook in geval AAAA van de stapel verdwijnt. Het effect op de stapel is dan bij alle uitvoeringsmogelijkheden gelijk:

```
uitslag      (n → )   Drukt afhankelijk van het getal op
                      de stapel de gewenste uitslag af.
```

Dit is een voorbeeld van waar je op moet letten bij IF-structuren:

Zorg er bij een IF-opdracht voor dat, zowel bij een conditie die 'false' is als bij een conditie die 'true' is, het effect op de stapel hetzelfde is.

Door je aan dit principe te houden kun je veel fouten voorkomen!

4.4 Logische operatoren voor gecombineerde condities

Vaak komen in een IF-opdracht 'samengestelde condities' voor. Deze condities ontstaan door enkelvoudige condities (als < en =) te combineren met behulp van logische operatoren als bijvoorbeeld AND en OR. Zo kunnen we in BASIC schrijven:

```
IF (X>10) AND (X<100) THEN .....
```

In FORTH zouden we hiervoor kunnen schrijven:

```
X @ 10   IF
  X @ 100   IF
              .....
            THEN
          THEN
```

maar een fraaiere en eenvoudiger oplossing maakt gebruik van {AND}:

```
X @ 10 > X @ 100 < AND IF ..... THEN
```

Hierin laat {X @ 10 >}een logische waarde op de stapel achter en evenzo {X @ 100 <}; deze worden door {AND} gecombineerd tot een nieuwe logische waarde op de stapel, die alleen dan true is wanneer de twee condities true hadden opgeleverd. De {AND}-opdracht is namelijk gedefinieerd met:

```
AND              (n1 n2 → n3)        n3 = n1 AND n2

false AND false = false
false AND true  = false
true  AND false = false
true  AND true  = true
```

De opdracht {AND} is in het voorgaande wel gebruikt om twee logische waarden met elkaar te combineren, maar is algemener in die zin dat daarmee ook twee getallen (op de stapel) van 16 bits bit voor bit met elkaar gecombineerd kunnen worden. Dit geldt trouwens ook voor de later te bespreken logische opdrachten {OR} en {XOR}. Bijvoorbeeld:

```
5 ok              (binair 101)
6 ok              (binair 110)
AND . 4 ok        (binair 100)
```

Terugkomend op het IF voorbeeld, moet opgemerkt worden dat het de variabele X is, die vergeleken wordt met 10 en 100. Dit maakt het eenvoudig omdat die X steeds met {X @} opgehaald kan worden. Het zal vaak gebeuren dat een getal op de stapel vergeleken moet worden met andere getallen, hetgeen dan enige stapel manipulatie vereist. Moet bijvoorbeeld een getal op de stapel met 0 en 100 vergeleken worden, dan kan dit met:

```
DUP 0< SWAP 100 > OR IF ....
```

Hierin is {0<} geen drukfout, maar een plezierige (en meestal snellere) versie van het uit twee woorden bestaande {0 <}. Het vergelijken met 0 komt zo vaak voor dat FORTH daarvoor de kortere schrijfwijze toelaat.

We kunnen het effect van de laatste reeks opdrachten nagaan met behulp van de in het vorige hoofdstuk besproken methode:

Woord	*Effect op stapel*	*Commentaar*
DUP	(n → n n)	dupliceer getal op de stapel
0<	(n n → n b1)	b1 is true als n negatief is
SWAP	(n b1 → b1 n)	verwissel n en b1
100	(b1 n → b1 n 100)	zet 100 op de stapel
>	(b1 n 100 → b1 b2)	b2 is true als n>100
OR	(b1 b2 → b3)	OR de twee b-waarden

Essentieel in deze reeks opdrachten zijn {DUP}, waardoor de waarde van n twee keer voorkomt - een voor iedere vergelijking -, en {SWAP}, die de twee bovenste stapelwaarden verwisselt zodat b1 bewaard wordt, terwijl de tweede vergelijking plaats vindt. Omdat {OR} alleen dan 'false' oplevert wanneer zijn beide operanden 'false' zijn, zal b3 'true' worden als b1 en/of b2 'true' is, m.a.w. als het oorspronkelijke getal op de stapel negatief of groter dan 100 was.

4.5 Ontbrekende vergelijkings-operaties

FORTH kent slechts drie vergelijkingsoperaties: {<}, {=} en {>}; dus niet de in andere talen voorkomende 'niet groter dan', 'niet kleiner dan' en 'niet gelijk aan' operaties. Deze komen niet in het standaard systeem voor, maar zijn gemakkelijk zelf te definiëren met behulp van de logische {NOT}, die van 'false' juist 'true' maakt en omgekeerd. Bijvoorbeeld:

```
: <= > NOT ;
: ≠  = NOT ;
: >= < NOT ;
```

In standaard FORTH is het Woordenboek niet volgestopt met alle mogelijke opdrachten; minder vaak gebruikte maar eenvoudige opdrachten als bovenstaande kan een gebruiker er zelf bij doen als hij daar behoefte aan heeft.

Opmerking. Je kunt je afvragen waarom een eenvoudige opdracht als {0<}wel in het standaard systeem zit. Het antwoord is dat in feite {0<} een 'primitievere' bewerking is dan {<}, die gedefinieerd is met

```
: < - 0< ;
```

Een andere nuttige vergelijkingsoperatie is die voor 16 bits getallen zonder teken {U<}. Bijvoorbeeld:

```
1 60000 U< . 1 ok          (dus true)
50000 40000 U< . 0 ok      (dus false)
```

Aardig is ook de toepassing

```
100 U<
```

die het (korte) equivalent is van

```
DUP 0< NOT SWAP 100 < AND
```

en onderzoekt of het getal op de stapel in het gebied 0 tot 99 ligt. Hierbij is er gebruik van gemaakt dat negatieve getallen blijkbaar grote positieve getallen zijn wanneer je de eerste bit niet als een teken-bit opvat.

Een nuttige stapel-manipulatie en vergelijkings-operatie is tenslotte {?DUP}, die een getal op de top van de stapel alleen dan dupliceert als dat getal niet-nul (of 'true') is. Gewoonlijk wordt deze opdracht vóór een 'IF' gebruikt, zodat wanneer het getal op de stapel niet-nul is, het wordt gedupliceerd voor gebruik in de IF-structuur. Is het getal nul, dan wordt het van de stapel afgehaald. Bijvoorbeeld:

```
: voorbeeld ?DUP IF ." top-getal is" . THEN ; ok
34 voorbeeld top-getal is 34 ok
0 voorbeeld ok
```

4.6 Samenvatting en oefeningen

In dit hoofdstuk zijn de volgende nieuwe opdrachten ingevoerd:

Stapel manipulatie

?DUP (n → n) of (n → n n)
n wordt alleen gedupliceerd als n niet-nul is.

Vergelijkingsbewerkingen

< (n1 n2 → b)
b is true als n1 (algebraisch) kleiner is dan n2.

`=` (n1 n2 → b)
b is true als n1 gelijk is aan n2.

`>` (n1 n2 → b)
b is true als n1 (algebraisch) groter is dan n2.

`0<` (n → b)
b is true als n kleiner dan nul (negatief) is.

`0=` (n → b)
b is true als n nul was.

`0>` (n → b)
b is true als n groter dan nul (positief) was.

`U<` (un1 un2 → b)
b is true als van de 16 bits getallen zonder teken `un1 < un2`.

`NOT` (b1 → b2)
Maak van 'false' 'true' en omgekeerd.

Logische bewerkingen

`AND`, `OR` en `XOR` (n1 n2 → n3)
Bit voor bit worden de volgende bewerkingen uitgevoerd:

```
0 AND 0 = 0      0 OR 0 = 0      0 XOR 0 = 0
0 AND 1 = 0      0 OR 1 = 1      0 XOR 1 = 1
1 AND 0 = 0      1 OR 0 = 1      1 XOR 0 = 1
1 AND 1 = 1      1 OR 1 = 1      1 XOR 1 = 0
```

Besturings-structuren

`IF` (b →)
Alleen te gebruiken in een colon-definitie als volgt:
`b IF ... ELSE ... THEN` of `b IF ... THEN`
Als `b` true is worden de opdrachten na `IF` uitgevoerd en de opdrachten na `ELSE` worden overgeslagen. Is `b` false, dan worden de opdrachten tussen `IF` en `ELSE` overgeslagen en de opdrachten na `ELSE` uitgevoerd. De uit te voeren opdrachten kunnen weer besturingsstructuren zijn.

Oefeningen

1) Laat zien wat het effect op de stapel is van:

```
1 2 >
-4 0<
5 0> NOT
```

2) Definieer de opdracht `{teken}`, die afhankelijk van het teken van het getal op de top van de stapel, een van de boodschappen 'positief', 'nul' of 'negatief' zal afdrukken.

3) Wat zal, binair geschreven, het resultaat zijn van de logische opdrachten:

```
1101101 1010001 XOR
1010 101 OR
4 5 = 2 3 < OR
```

4) Hoe zou je de volgende BASIC IF-opdracht in FORTH schrijven, aannemend dat de variabelen A en B al gedefinieerd zijn:

```
IF NOT((A=2) AND (B=2)) THEN LET A=4
```

5) Hebben de FORTH woorden {NOT} en {0=} iets gemeen?

6) Wat is het effect van uitvoering van de volgende colon-definities:

```
: vb1 OVER OVER > IF SWAP THEN DROP ;
: vb2 DUP IF DUP THEN ;
```

5 FORTH STRUCTUREN 2: LUSSEN

In de inleiding van het vorige hoofdstuk werd al vermeld dat de derde voorwaarde voor gestructureerd programmeren bestaat uit het herhaald (in 'lussen') kunnen uitvoeren van een reeks bewerkingen. In FORTH zijn drie lusstructuren mogelijk: de DO-lus, de UNTIL-lus en de WHILE-lus. Deze drie structuren worden in dit hoofdstuk besproken, waarna beschreven wordt hoe 'geneste' structuren worden gemaakt en op fouten kunnen worden onderzocht.

5.1 De DO-lus

De eenvoudigste (en meest gebruikte) van de drie lusstructuren is in FORTH de DO-lus. Deze wordt gebruikt als we vooraf weten, of kunnen berekenen hoe vaak een lus herhaald moet worden. Net als de IF-structuur mogen DO-lussen alleen in colon-definities voorkomen, zodat we de DO-lus zullen toelichten met een eenvoudige definitie:

```
: onderstreep CR 16 0 DO ." _" LOOP ; ok
onderstreep
________________ ok
```

De twee getallen 16 en 0 voor het woord {DO} geven aan hoe vaak de DO-lus moet worden uitgevoerd. Het eerste getal heet de 'limiet', het tweede de 'index'. De lus wordt (limiet - index) keer uitgevoerd, dus 16 - 0= 16 maal in dit geval waardoor {." _"} 16 maal wordt verricht met het vermelde resultaat.

De structuur van de DO-lus kan als volgt samengevat worden:

```
limiet  index  DO  ....FORTH opdrachten.... LOOP
```

In déze DO-lus moet de limietwaarde altijd groter zijn dan de index-waarde en de 'FORTH opdrachten' worden altijd (limiet - index) maal uitgevoerd. Is de limietwaarde nièt groter dan de indexwaarde, dan worden de 'FORTH opdrachten' precies één keer uitgevoerd en de lus is afgelopen. Na afloop van de DO-lus worden de opdrachten achter {LOOP} uitgevoerd.

Aangezien de DO-opdracht zijn limiet en index van de stapel afhaalt, hoef je deze waarden niet expliciet mee te geven in een colon-definitie, maar kun je een van de twee of beiden parameters maken. In de praktijk zul je de indexwaarde en niet de limietwaarde in de colon-definitie opnemen. Bijvoorbeeld:

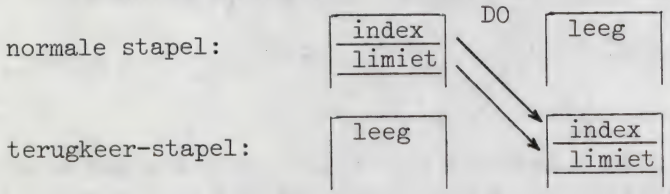
```
: kreten!  0 DO  CR ." O hemel!" LOOP ; ok
4 kreten!
O hemel!
O hemel!
O hemel!
O hemel! ok
```

5.2 De werking van de DO-lus

De DO-lus maakt voor zijn werking gebruik van een tweede, speciale, stapel (de 'terugkeer'-stapel) waarop de index- en limietwaarden worden bewaard. (De terugkeerstapel wordt door FORTH in hoofdzaak gebruikt tijdens de interpretatie-fase. Tijdens de vertaal-fase is deze stapel vrij beschikbaar, zodat de DO-lus er gebruik van kan maken. Ook de FORTH programmeur zou er in een colon-definitie profijt van kunnen hebben, maar zoals in paragraaf 8.3 besproken zal worden, moet dit met zorg gebeuren! De in vorige hoofdstukken steeds gebruikte stapel wordt ook wel 'gewone' stapel of 'parameter' stapel genoemd.)

Tijdens de uitvoering van een DO-lus gebeurt het volgende:

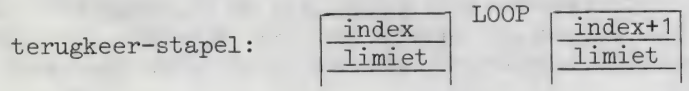
i) Het woord {DO} wordt slechts één keer 'uitgevoerd' en bewerkstelligt het overbrengen van de twee bovenste getallen op de gewone stapel naar de terugkeer-stapel, zoals in figuur 5.1 aangegeven:



Figuur 5.1 De werking van {DO}

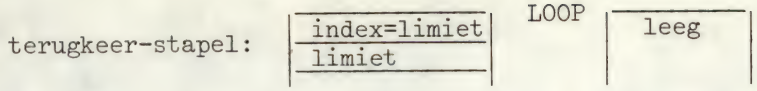
ii) De opdrachten in de DO-lus worden gewoon uitgevoerd.

iii) Door het woord {LOOP} wordt 1 opgeteld bij de indexwaarde op de terugkeer-stapel, die vervolgens wordt vergeleken met de limietwaarde daaronder. {LOOP} doet niets met de gewone stapel. Is de nieuwe indexwaarde kleiner dan de limietwaarde, dan worden de opdrachten achter {DO} opnieuw uitgevoerd.



Figuur 5.2 Effect van {LOOP}, *index+1<limiet*

Is daarentegen de nieuwe indexwaarde gelijk aan de limietwaarde, dan wordt de terugkeer-stapel leeg gemaakt en de opdracht na {LOOP} wordt uitgevoerd.



Figuur 5.3 Effect van het einde van de lus

Een DO-lus zal uiteindelijk dus de toestand van de terugkeer-stapel niet veranderen; m.a.w. zoals hier aangenomen leeg laten. De voorgaande beschrijving van de werking van een DO-lus mag ingewikkeld lijken, maar de programmeur hoeft er zich niet veel van aan te trekken omdat de DO-lus voor zichzelf zorgt.

5.3 Berekeningen in een lus

Het ligt voor de hand dat we in de meeste toepassingen graag gebruik zullen willen maken van de indexwaarde, die in elke doorloop van de lus steeds 1 groter wordt. Dit nu is mogelijk met de opdracht {I}, die een kopie van de momentane indexwaarde op de terugkeerstapel bovenop de normale stapel zet. Bijvoorbeeld:

```
: kwadraten 0 DO I I * . LOOP ; ok
10 kwadraten 0 1 4 9 16 25 36 49 64 81 ok
```

Wederom is het leerzaam om de werking van {kwadraten} te bestuderen met behulp van de stapel-notatie (alleen voor de normale stapel):

Woord	*Effect op stapel*	*Toelichting*
0	(n^ → n 0)	zet index op 0
DO	(n 0 →)	lus gaat van 0 tot n
I	(† → i)	kopieer index
I	(i → i i)	kopieer index nogmaals
*	(i i → i*i)	bereken kwadraat
.	(i*i → †)	en druk het af
LOOP	(→ ^)	einde lus?

De hierin met ^ aangegeven plaatsen geven het totale effect van:

```
kwadraten     (n → )     Druk de kwadraten van 0 t/m n-1 af.
```

Merk op dat de stapel leeg is op de met † aangegeven plaatsen. Het is essentieel dat iedere herhaling van de lus geen getallen van de stapel verwijdert of toevoegt omdat anders STACK EMPTY of STACK FULL foutmeldingen het gevolg kunnen zijn van een vaak herhaalde lus! Het is daarom nuttig om op de met † aangeduide plaatsen te controleren dat het aantal getallen op de stapel niet veranderd is (heel soms zijn er uitzonderingen op deze aanbeveling).

De DO-lus en de BASIC FOR opdracht lijken veel op elkaar, zoals uit figuur 5.4 blijkt. Merk echter op dat de limietwaarden niet gelijk zijn. De BASIC lus wordt uitgevoerd voor A van 0 t/m 9, maar in FORTH moet de limietwaarde gezet worden op 9+1=10.

```
BASIC                          FORTH

                               : kwadraten
10 FOR A=0 TO 9                  10 0 DO
20 PRINT A*A                           I I * .
30 NEXT A                            LOOP
                               ;
```

Figuur 5.4 BASIC FOR en FORTH DO

5.4 Een variant van de DO-lus

Met het woord {+LOOP} kun je een zeer welkome verfijning van de DO-lus realiseren, nl. het telkens verhogen van de index met andere waarden dan +1. Het woord {+LOOP} wordt in plaats van {LOOP} gebruikt en het enige verschil is eigenlijk dat het eerste woord het bovenste getal van de stapel afhaalt en optelt bij de index alvorens na te gaan of de lus afgelopen is of niet. Als de index bijvoorbeeld achtereenvolgens de waarden 3, 6, 9, 12 en 15 moet aannemen, is een geschikte DO-lus:

```
16 3 DO ..... 3 +LOOP
```

Als we evenzo de index telkens willen verlagen, bijvoorbeeld in de reeks 10, 5, 0, -5, -10, dan kan dat met

```
-11 10 DO ..... -5 +LOOP
```

Merk op dat de lus afgelopen is wanneer de indexwaarde gelijk is aan de limietwaarde of deze passeert.

Vanzelfsprekend kan de 'stapwaarde' een getal zijn dat in de lus zelf berekend wordt, zoals we in het volgende voorbeeld zullen zien:

```
: voorbeeld 100 1 DO I . I +LOOP ; ok
voorbeeld 1 2 4 8 16 32 64 ok
```

Dit geeft een fraaiere manier om een stel waarden af te drukken die anders berekend zouden moeten worden.

5.5 Nesten van DO-lussen en andere bijzonderheden

Net als andere structuren mogen DO-lussen binnen dezelfde colon-definitie genest zijn in andere DO- en IF-structuren. Laten we echter eerst geneste DO-lussen bekijken:

```
: *tabel CR 11 1 DO 11 1 DO J I * . LOOP CR LOOP ; ok
*tabel
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 ....enz.
```

Het hier gedefinieerde woord {*tabel} heeft bij uitvoering het effect dat een vermenigvuldigingstabel wordt afgedrukt (zie voor een fraaier afdruk het woord {.R} in paragraaf 8.4).

De werking van deze opdracht berust op het woord {J} in de binnenlus, dat net zoiets is als {I} behalve dat {J} in de binnenste lus de index van de buitenste lus op de stapel plaatst. (We hoeven het niet te vertellen, maar {J} zat onder de limietwaarde op de terugkeer-stapel.) Voor elke herhaling van de buitenlus, waarin J achtereenvolgens van 1 naar 10 gaat, wordt de binnenlus 10 keer uitgevoerd met I van 1 oplopend naar 10. Vermenigvuldiging van J en I geeft dus 1*1, 1*2, 1*3,....1*10 en dan 2*1, 2*2, ... 2*10, en zo door tot 10*10; m.a.w. de vermenigvuldigingstabel van de gehele getallen tot 10. Om het nog eens toe te lichten, volgt wederom het woord {*tabel}, samen met een equivalent BASIC programma waarin met opzet voor de lusvariabelen de letters J en I gekozen zijn:

```
BASIC                          FORTH

                               : *tabel
10 FOR J=1 TO 10                   11 1 DO
20   FOR I=1 TO 10                       11 1 DO
30     PRINT J*I                             J I * .
40   NEXT I                                  LOOP
50   PRINT                                   CR
60 NEXT J                              LOOP ;
```

Figuur 5.5 Geneste DO-lussen in BASIC en FORTH

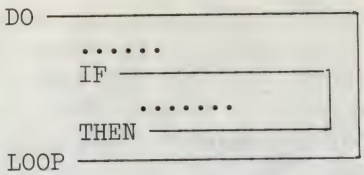
Onthoud echter goed dat {J} en {I} in FORTH géén variabelen zijn, al worden ze wel als zodanig gebruikt in dit voorbeeld!

Een ander woord dat uitsluitend in een DO-lus gebruikt wordt, is {LEAVE} dat het mogelijk maakt om een lus voortijdig te beëindigen. Het effect van dit woord is eenvoudigweg dat de limietwaarde gelijk wordt gemaakt aan de momentane indexwaarde. Als daarna {LOOP} of {+LOOP} tegengekomen wordt zal de lus niet meer doorlopen worden. Gewoonlijk wordt {LEAVE} in een IF-structuur binnen een DO-lus gebruikt, zoals in:

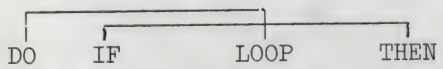
```
: voorbeeld 10000 1 DO I .
                       ?TERMINAL IF LEAVE THEN
                    LOOP ;
```

In dit voorbeeld is aangenomen dat eerder al een woord {?TERMINAL} is gedefinieerd met het effect na te gaan of op het toetsenbord een toets is aangeslagen. Zo ja, dan wordt 'true' op de stapel geplaatst, en anders 'false'. (De meeste FORTH-systemen kennen dit woord, al hoort het niet tot de FORTH-79 norm.) Als {voorbeeld} bij het uitvoeren niet gestoord wordt, dan worden de getallen 1 t/m 9999 afgedrukt, maar dit proces kun je op ieder ogenblik afbreken door een willekeurige toets op het toetsenbord aan te slaan. Iets dergelijks kan handig zijn wanneer je een programma voortijdig wilt laten beëindigen!

Alvorens dit voorbeeld te verlaten, is het nuttig om nog eens te letten op de gedaante van een geneste IF-structuur in een DO-lus:



Verbinden we IF en THEN en evenzo DO en LOOP door lijnen, dan zien we dat de IF-structuur geheel omsloten wordt door de DO-lus, zodat de structuur van het geheel correct is. Als de lijnen elkaar kruisen, dan is de structuur zeker verkeerd en zal het geheel niet correct werken, zoals aangegeven in figuur 5.6:



Figuur 5.6 Een niet-correcte geneste structuur

Wanneer je ooit twijfelt over de correctheid van geneste structuren, gebruik dan deze 'verbindingstest' door corresponderende DO .. LOOP, IF .. THEN (en latere BEGIN .. UNTIL en BEGIN .. REPEAT) woorden met elkaar te verbinden. Kruisen de lijnen elkaar ergens, dan is er zo goed als zeker iets mis met de structuur.

5.6 De UNTIL-lus

De DO-lus wordt vaak een lus met 'vaste grenzen' genoemd omdat het aantal herhalingen al (afgezien van de {LEAVE}-mogelijkheid) 'vast staat' voordat de lus wordt uitgevoerd. De UNTIL- en WHILE-lus hebben geen 'vast' aantal herhalingen omdat het einde van de lus pas tijdens de uitvoering ervan vastgesteld wordt op grond van een berekening in de lus zelf.

De gedaante van een UNTIL-lus is in het algemeen:

```
BEGIN ...FORTH opdrachten... conditie UNTIL
```

waarin de FORTH opdrachten steeds opnieuw worden uitgevoerd totdat de 'conditie' true oplevert. In die conditie zullen gewoonlijk vergelijkingsoperaties staan, die een logische waarde op de top van de stapel moeten achterlaten voor {UNTIL}. Een voorbeeld van een UNTIL-lus is:

```
: wacht BEGIN KEY 32 = UNTIL ;
```

Hierin is het nieuwe woord {KEY} ingevoerd, dat het effect heeft de uitvoering van het programma te onderbreken ('te wachten') totdat een toets op het toetsenbord is aangeslagen en alsdan de ASCII-waarde van die toets op de top van de stapel te plaatsen:

```
KEY        ( → kar)    Wacht totdat een toets is aangeslagen
                       en zet dan de waarde ervan op de stapel.
```

Het 'conditiedeel' van de UNTIL-lus onderzoekt of de waarde van de aangeslagen toets soms 32 (voor een spatie) is en herhaalt de lus zolang dat niet het geval is. Het totale effect van {wacht} is dus dat met de verdere verwerking van het programma gewacht wordt totdat de spatie-toets op het toetsenbord is ingedrukt. In detail bekeken:

Woord	*Effect op stapel*	*Toelichting*
BEGIN	(→)	begin van de lus
KEY	(→ kar)	karakter van toetsenbord gelezen
32	(kar → kar 32)	32 is de ASCII-waarde van spatie
=	(kar 32 → b)	b wordt true als kar=32
UNTIL	(b →)	terug naar BEGIN als b=false

5.7 De WHILE-lus

De gedaante van een WHILE-lus is iets ingewikkelder dan die van een UNTIL-lus en is in het algemeen:

```
BEGIN conditie WHILE ... FORTH opdrachten... REPEAT
```

Is de conditie true dan worden de FORTH opdrachten uitgevoerd en er wordt opnieuw bij BEGIN naar de conditie gekeken; is de conditie (aanvankelijk of later) false dan worden de FORTH opdrachten niet (meer) uitgevoerd en is de lus afgelopen. Anders gezegd: alles tussen BEGIN en REPEAT wordt herhaaldelijk uitgevoerd zolang de conditie true is.

Vaak kan zowel een WHILE-lus als een UNTIL-lus gebruikt worden. De definitie van {wacht} kun je bijvoorbeeld ook met een WHILE-lus schrijven:

```
: wacht BEGIN KEY 32 = NOT WHILE ( doe niets ) REPEAT ;
```

waarbij je erom moet denken de conditie andersom te gebruiken; de lus wordt herhaald zolang de aangeslagen toets *niet* de spatietoets is.

Hoewel de WHILE- en UNTIL-lus bijna identiek lijken of zelfs als alternatief voor elkaar gebruikt mogen worden, is er toch een essentieel verschil tussen deze twee en wel dat de UNTIL-lus altijd tenminste één keer wordt uitgevoerd, terwijl dat niet het geval is voor de WHILE-lus. Die wordt namelijk niet uitgevoerd als de conditie aanvankelijk al false is! Op grond van dit verschil kan de FORTH programmeur dan de meest geschikte lus kiezen voor een gegeven probleem.

5.8 De werking van FORTH structuren

De volgorde van uitvoering van opdrachten in een colon-definitie wordt geheel bepaald door de structuren in die definitie. Omdat verder onvolledige structuren (IF zonder THEN, DO zonder LOOP, enz. en GOTO's niet toegestaan zijn, is het als regel gemakkelijk om de volgorde van gebeurtenissen na te gaan wanneer een colon-definitie uitgevoerd wordt. Laten we bijvoorbeeld de volgende definitie onderzoeken:

```
: afdruk DUP 0> IF 0 DO ." *" LOOP
                 ELSE DROP ." -"
                THEN
                CR ;
```

Er zijn maar twee uitvoeringsmogelijkheden in {afdruk}; hetzij de 'true-opdrachten' worden uitgevoerd, hetzij de 'false-opdrachten' (afhankelijk van wat oorspronkelijk op de stapel stond), maar zeker niet beide. In de volgende figuur is dit nog eens geschetst:

```
                                    ---------> tenminste 1 keer
uitvoering 1:  DUP 0> (IF) 0 (DO) ." *" CR ;
uitvoering 2:  DUP 0> (IF) DROP ." -" CR ;
```

In deze figuur zijn de structuur-bepalende woorden niet vermeld, behalve (en dan tussen haakjes) die woorden die de (normale) stapel beïnvloeden. Merk op dat de uitvoering altijd begint met het eerste woord in de definitie, dus {DUP} hierboven, en altijd eindigt met de afsluitende kommapunt. Als {afdruk} opgenomen is in een andere definitie, bijvoorbeeld:

```
: test 11 -10 DO I afdruk LOOP ;
```

zal iedere keer dat {afdruk} 'aangeroepen' wordt tijdens uitvoering van {test} òf uitvoering 1 òf uitvoering 2 gekozen worden, maar de afsluitende kommapunt bewerkstelligt 'terugkeer' naar {test}.

Op deze manier zal ook een ingewikkeld programma met vele 'niveaus' van colon-definities op een nette en voorspelbare wijze verwerkt worden. Mits de colon-definities eenvoudig zijn, zal het controleren van de werking van een programma niet moeilijk zijn. Eenvoud is geboden: FORTH programmeurs zijn het er over eens dat een colon-definitie niet meer dan drie à vier structuren zou moeten bevatten.

Er zijn uitzonderingsgevallen waarin we de verwerking van een opdracht willen afbreken omdat zich een situatie voordoet die voortzetting van de verwerking onmogelijk of zinloos maakt. Hierin voorziet FORTH met de woorden {ABORT} en {QUIT}, die beiden de verdere verwerking afbreken en voortzetting van het werk door middel van het toetsenbord vereisen. Het eerste woord veegt alle stapels schoon en produceert gewoonlijk een boodschap, het tweede woord laat de normale stapel zoals die was en produceert geen boodschap. We zouden bijvoorbeeld de bewerking 'delen' kunnen herdefiniëren opdat die waarschuwt voor een deling door nul:

```
: / DUP                          ( dupliceer de deler )
      IF                         ( is die niet nul )
        /                        ( voer dan de deling uit )
       ELSE
        ." Deling door nul! " ABORT
      THEN ;
```

Deze nieuwe delingsbewerking gedraagt zich als de oude, behalve wanneer getracht zou worden door nul te delen. In dat geval wordt de

boodschap "Deling door nul!" afgedrukt en de verwerking stopt. Bij gebruik van het woord {QUIT} in plaats van {ABORT} blijft de normale stapel intact. Deze kan dan geanalyseerd worden om fouten op te sporen.

5.9 Samenvatting en oefeningen

In dit hoofdstuk zijn de volgende nieuwe opdrachten ingevoerd:

Besturings-structuren

DO (n1 n2 →)
Alleen te gebruiken in een colon-definitie als volgt:
DO ..... LOOP of DO ..... +LOOP
Zorgt voor een lus met vaste grenzen, bepaald door de limietwaarde n1 en de beginwaarde n2 van de index.

LOOP (→)
Zorgt voor het verhogen van de indexwaarde van een DO-lus met 1 en voor het beëindigen van de lus wanneer de index niet kleiner is dan de limiet.

+LOOP (n →)
Zorgt voor het verhogen van de indexwaarde met n en het vergelijken van de nieuwe indexwaarde met de limietwaarde. Beëindigt voor positieve n de lus wanneer de indexwaarde niet kleiner is dan de limietwaarde; voor negatieve n wanneer de indexwaarde kleiner is dan de limietwaarde.

I (→ n)
Indien gebruikt in DO ....I.... LOOP, zal dit de momentane indexwaarde op de stapel zetten.

J (→ n)
Indien gebruikt in DO .. DO .. J .. LOOP .. LOOP zal dit de momentane indexwaarde van de buitenste lus op de stapel zetten.

LEAVE (→)
Maakt de limietwaarde van een DO-lus gelijk aan de momentane indexwaarde, zodat de lus bij de volgende {LOOP} of {+LOOP} wordt beëindigd. De index verandert niet, zodat opdrachten tussen {LEAVE} en {LOOP} of {+LOOP} normaal worden uitgevoerd.

BEGIN (→)
Staat aan het begin van een UNTIL- of WHILE-lus en is alleen in een colon-definitie te gebruiken als volgt:
BEGIN ..... UNTIL BEGIN ..... WHILE .... REPEAT
Er moet voor worden gezorgd dat voor {UNTIL} of {WHILE} een logische waarde ('true' of 'false') op de top van de stapel staat.

UNTIL (b →)
Is die logische waarde false, dan wordt de lus herhaald bij de opdracht na {BEGIN}. De lus is afgelopen als b true is .

WHILE (b →)
Is voor {WHILE} de logische waarde b true, dan worden de opdrachten tussen {WHILE } en{REPEAT} uitgevoerd en daarna de opdrachten na {BEGIN}. Is (wordt) b false, dan is de lus afgelopen en wordt de opdracht na {REPEAT} uitgevoerd.

REPEAT (→)
Staat aan het einde van een BEGIN .. WHILE .. REPEAT-lus.

Invoer en diversen

KEY (→ kar)
Wacht op het aanslaan van een toets en plaatst dan de ASCII-waarde daarvan op de stapel.

ABORT (n1 n2 .... →)
Veegt alle stapels schoon en wacht op voortzetting van het werk met behulp van het toetsenbord.

QUIT (→)
Veegt de terugkeer-stapel schoon en wacht op voortzetting van het werkmet behulp van het toetsenbord.

Oefeningen

1) Definieer een opdracht die het effect heeft dat een blok sterretjes wordt afgedrukt, zodat bijvoorbeeld het volgende gebeurt:

```
4 sterren
****
****
****
****
ok
```

2) Schrijf een programma om een rij gehele getallen bij elkaar op te tellen, bijvoorbeeld:

```
1 10 sigma
```

3) Schrijf een programma om van een zekere startwaarde af t/m 0 de desbetreffende waarden af te drukken met telkens ongeveer een seconde tussenpauze. Druk aan het einde een passende boodschap af; bijvoorbeeld: "We zijn gestart".

4) Wat zal door de volgende lussen afgedrukt worden:

```
: vb1 16 0 DO I . 3 +LOOP ;
: vb2 0 10 DO I . -1 +LOOP ;
: vb3 5 BEGIN DUP . 5 + DUP 100 > UNTIL . ;
```

5) Schrijf een programma dat alleen die getallen tussen twee gegeven waarden afdrukt, die deelbaar zijn door een derde gegeven getal.

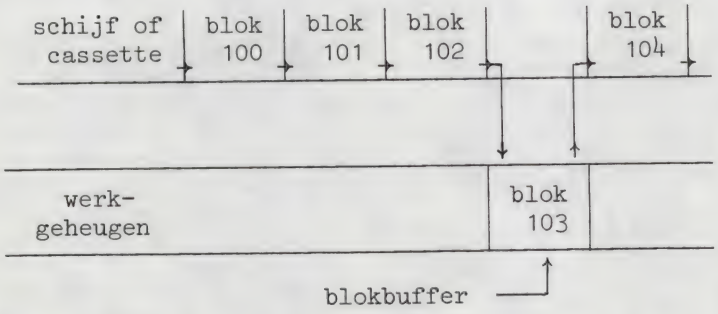
6) Schrijf een {DUMP}-programma dat vanaf een gegeven adres de inhoud van het geheugen zal afdrukken in blokken van 8 regels van ieder 8 bytes. Aan het einde van ieder blok moet het programma wachten op het aanslaan van een toets; is dat een spatie, ga dan door met afdrukken en beëindig anders het programma. (Aanwijzing: je zult twee geneste DO-lussen in een UNTIL-lus nodig hebben en bovendien de {.R}- opdracht uit paragraaf 8.4.)

6 HET OPMAKEN, BEWAREN EN LADEN VAN PROGRAMMA'S

We zijn nu zover dat we voldoende FORTH opdrachten hebben behandeld om praktisch bruikbare en tamelijk ingewikkelde programma's te maken. De tot dusver gebruikte methode van het direct invoeren van een programma (met behulp van het toetsenbord) is echter wel bruikbaar voor kleine oefenvoorbeelden, maar niet voor het ontwerpen van ingewikkelde programma's. We hebben echt behoefte aan methoden om een programma op te maken, op schijf of cassetteband op te slaan en van schijf of cassetteband in te lezen (te laden), en wel in dezelfde vorm als ware het direct ingetypt. FORTH voorziet in deze behoefte door de schijf en de cassetteband te beschouwen als een verlengstuk van het interne computergeheugen. Informatici noemen dit wel een 'virtueel geheugen' en in de praktijk betekent dit dat het oorspronkelijke FORTH programma erg groot mag zijn, zonder dat het veel geheugenruimte vergt.

6.1 Het FORTH LAAD-concept

FORTH verdeelt de ruimte op schijf of cassette in 'blokken' van 1024 karakters. Blokken worden een voor een geladen in 'blokbuffers' in het werkgeheugen, waar ze 'opgemaakt' of uitgevoerd kunnen worden. Programma's kunnen echter meer blokken beslaan, en in een blok kunnen opdrachten staan om andere blokken te 'laden' (zodat de programmeur dit niet via het toetsenbord hoeft te doen).



Figuur 6.1 Het FORTH LAAD-concept

In de praktijk kan het gebeuren dat we zojuist opgemaakte blokken willen redden, bijvoorbeeld voordat we schijven gaan verwisselen of voordat we de machine uitzetten of als voorzorgsmaatregel voordat we nieuwe (of riskante) definities gaan testen. FORTH kent daarvoor een 'SAVE-BUFFERS' opdracht, die later in dit hoofdstuk besproken wordt.

6.2 De Editor

Een 'editor' is een onderdeel van een FORTH-systeem dat het 'opmaken' van een programma vereenvoudigt. In de FORTH-79 norm zijn geen editor opdrachten gespecificeerd, zodat deze opdrachten van systeem tot systeem verschillend kunnen zijn. We zullen hier een klein aantal 'typische' editor opdrachten (afkomstig van een FORTH groep) bespreken, maar lezers met een eigen systeem moeten, terwijl ze dit doornemen, de documentatie van hun systeem bestuderen voor meer gedetailleerde informatie.

Bij veel FORTH-systemen kun je de editor woordenschat (d.i. de verzameling van editor opdrachten) niet bereiken voordat je een aantal blokken van schijf of cassette geladen hebt, of bij sommige systemen het woord EDITOR hebt ingetikt. Wederom, raadpleeg je eigen systeem documentatie om toegang te krijgen tot de editor woorden.

Laten we eens aannemen dat we een pas ontworpen FORTH programma in een blok willen plaatsen. We moeten dan eerst een leeg of niet meer benodigd blok vinden (bij sommige cassette systemen is dit niet nodig). De veiligste manier om te controleren of een blok daarvoor in aanmerking komt, is het afdrukken van een blok; bijvoorbeeld:

```
100 LIST
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
ok
```

Het invoeren van nieuwe tekst is meestal het eerste dat we willen doen. Dit kan regel voor regel gebeuren met het editor woord {P} (te onthouden met het woord 'plaatsen'). Het intikken van een regelnummer (van 0 t/m 15), gevolgd door {P}, een tekst van ten hoogste 64 karakters en het aanslaan van de 'return'-toets heeft tot resultaat dat de tekst op de gewenste regel geplaatst wordt. (Stond op die regel oorspronkelijk iets anders dan wordt dit vervangen door de nieuwe tekst.) Voorbeeld:

```
0 P ( Testblok ) ok
2 P : kwadraten          ( druk kwadraten van 0 t/m 9 af ) ok
3 P 10 0 DO ok
4 P       I I * . ok
5 P      LOOP ; ok
```

Nadat we zo enkele regels ingetikt hebben, zullen we waarschijnlijk dit blok opnieuw willen afdrukken om te controleren of we geen fouten hebben gemaakt. Dit kan met {100 LIST}, maar nog eenvoudiger met het editor woord {L}, dat tot resultaat heeft dat de momenteel gebruikte blokbuffer afgedrukt wordt:

```
L
 0 ( Testblok )
 1
 2 : kwadraten            ( druk kwadraten van 0 t/m 9 af )
 3 10 0 DO
 4       I I * .
 5      LOOP ;
 6
 7
 8
 9
10
11
12
13
14
15
ok
```

Als we slechts één enkele regel in de buffer willen zien, kunnen we het woord {T} (denk aan 'typen') gebruiken. Bijvoorbeeld:

```
2 T
2 : kwadraten            ( druk kwadraten van 0 t/m 9 af )
ok
```

Het toevoegen van een nieuwe regel tussen twee oude regels van de buffer vergt twee opdrachten. Eerst {S}, dat de tekst in de buffer 'spreidt' en een blanko regel toevoegt, en dan {P} voor de nieuwe regel.

```
2 S ok
2 P kwadraten              ( probeer kwadraten ) ok
```

zal bewerkstelligen dat de definitie van 'kwadraten' op regel 3 begint. De omgekeerde bewerking: het doorhalen van een regel door de daaronderstaande regels omhoog te schuiven, gebeurt met het editor woord { D}.

We kunnen tenslotte een hele regel van de ene plaats in de buffer naar een andere overbrengen met behulp van een 'kladgeheugen' van één regel, dat de naam PAD heeft. De 'doorhaal' opdracht {D} zet de weggehaalde regel namelijk in PAD, vanwaar het naar een andere regel gekopieerd kan worden met de 'invoeg' opdracht {I}, die eerst de buffertekst 'spreidt' op de aangegeven plaats en vervolgens de zo ontstane blanko regel vervangt door de tekst uit PAD. Om bijvoorbeeld de net gemaakte nieuwe regel 2 te verplaatsen naar regel 8, tikken we:

```
2 D ok
8 I ok
```

en een volledige afdruk van het blok geeft nu:

```
L
 0 ( Testblok )
 1
 2 : kwadraten              ( druk kwadraten van 0 t/m 9 af )
 3 10 0 DO
 4       I I * .
 5      LOOP ;
 6
 7
 8 kwadraten                ( probeer kwadraten )
 9
10
11
12
13
14
15
ok
```

Een samenvatting van de tot nu toe behandelde editor woorden is:

```
P tekst (n→ ) Plaats tekst (afgesloten met 'return') op regel n
L       ( → ) Lijst de in gebruik zijnde blokbuffer af
T       (n→ ) Tik regel n uit en zet die ook in PAD
S       (n→ ) Spreid de buffertekst, zodat regel n blanko wordt
D       (n→ ) Doorhalen van regel n (na kopiëren in PAD) door de
              daaronder staande regels omhoog te schuiven
I       (n→ ) Invoegen van de regel uit PAD op regel n na sprei-
              ding van de buffertekst op die plaats
PAD     ( →adres) Zet het adres van PAD op de stapel (FORTH-79)
```

Deze editor woorden zijn al voldoende om een FORTH programma in te voeren en regel voor regel te bewerken. Er zijn nog wel meer editor woorden voorgesteld, waarmee een nog geraffineerder opmaak mogelijk is, zoals bijvoorbeeld het veranderen van slechts een deel van een regel (zonder dus de hele regel opnieuw in te moeten tikken).

Willen we het programma in blok 100 testen, dan tikken we:

```
100 LOAD
0 1 4 9 16 25 36 49 64 81 ok
```

en blijkens de uitvoer is {kwadraten} correct vertaald en uitgevoerd.

Opmerking. Bij sommige systemen die van een cassette gebruik maken zal een LOAD-opdracht altijd een blok van de band lezen, ongeacht of dat blok al in de buffer was of niet. In dat geval bestaan meestal opdrachten als ENTER of EXEC om een al aanwezig blok te 'laden'.

Wanneer we daarentegen een tikfout zouden hebben gemaakt in de definitie van {kwadraten}, bijvoorbeeld 'LLOP' in plaats van 'LOOP' in regel 5:

```
100 LOAD
LLOP ?
```

dan zal FORTH het (in Amerika!) aanstoot gevende woord afdrukken, samen met de foutmelding '?', die betekent dat zo'n woord niet in het Woordenboek stond. De verwerking wordt dan afgebroken en via het toetsenbord moeten we dan het blok en de daarin staande fout opzoeken, de fout corrigeren en het blok opnieuw laden.

Zodra een blok helemaal correct is, zullen we het willen bewaren op schijf of cassette. Dit gebeurt automatisch wanneer we door zouden gaan met het afdrukken en opmaken van andere blokken. Immers als onze blokbuffer nodig is voor een ander blok, wordt blok 100 eerst teruggeschreven naar schijf of cassette. Zijn we voorlopig echter klaar met ons werk, dan kunnen we blok 100 in veiligheid brengen met:

```
SAVE-BUFFERS
```

Beginnelingen in FORTH wordt aangeraden deze opdracht geregeld te gebruiken totdat ze helemaal vertrouwd zijn met het 'spelen' met blokken.

6.3 Nog meer blok-bewerkingen

De drie bewerkingen LIST, LOAD en SAVE-BUFFERS zijn, samen met wat editor woorden, voldoende om allerlei toepassingsprogramma's op schijf of cassette te zetten. FORTH voorziet echter in nog meer bewerkingen met blokken, waarmee je o.a. met een programma blokken heen en weer kunt sturen tussen het werkgeheugen en schijf of cassette. Daarmee kun je bijvoorbeeld 'gegevensblokken' lezen of vastleggen. Dit wordt in deze paragraaf besproken, maar is niet essentieel voor beginners.

Figuur 6.1 illustreert dit concept; verondersteld is dat een groot programma de blokken 100 en volgende beslaat. Om het hele programma te laden (het bestaat bijna zeker uit een groot aantal colon-definities), hoeft de FORTH programmeur slechts in te tikken:

```
100 LOAD
```

De opdracht {LOAD} betekent: 'kopieer het blok, waarvan het nummer op de top van de stapel staat, in een blokbuffer in het werkgeheugen en laat het vervolgens interpreteren alsof het rechtstreeks ingetikt was'. Mogelijke colon-definities in het blok worden dus vertaald en gewoon toegevoegd aan het Woordenboek. Als de opdracht {101 LOAD} aan het einde van blok 100 staat, wordt blok 101 onmiddellijk na blok 100 geladen in dezelfde blokbuffer (en wordt dus niet gewacht tot het werk wordt voortgezet via het toetsenbord). Een willekeurig aantal blokken kan zo 'geketend' worden, zodat een groot programma geheel geladen kan worden zonder meer dan 1 Kbytes werkgeheugen (voor de blokbuffer) nodig te hebben. Figuur 6.1 laat zien dat blok 103 net geladen is.

Het FORTH systeem behandelt ieder blok alsof het bestaat uit 16 regels van ieder 64 karakters (een geschikte maat voor een beeldscherm), zodat de 5 in figuur 6.1 weergegeven blokken een programma van 80 regels kunnen bevatten. (Sommige FORTH systemen spreken daarom over een 'scherm' in plaats van over een blok, zodat deze twee termen door elkaar gebruikt mogen worden.) Over wat een blok mag bevatten, bestaan geen vaste voorschriften. Alles wat je direct in kunt tikken (en dat is onbegrensd!), kan opgemaakt worden tot een blok op schijf of cassette. Ons programma in de blokken 100 t/m 104 bestaat waarschijnlijk uit een aantal colon-definities, declaraties van variabelen en constanten, veel commentaar en wat FORTH opdrachten, die direct tijdens het laden uitgevoerd moeten worden.

Op de volgende pagina volgt een voorbeeld van een, op papier afgedrukt, verzonnen blok (de opdracht {LIST} kopieert een blok en maakt er een lijst met 16 genummerde regels van op de schrijfmachine). Afspraak is dat regel 0 bestaat uit commentaar dat de inhoud van het blok beschrijft. De twee colon-definities in dit blok, op regels 2-5 en 8-14, zijn ingesprongen voor een betere leesbaarheid. Regels 6 en 15 zorgen voor de uitvoering van deze colon-definities (om ze te testen) nadat dit blok geladen is.

Iedere regel in het blok wordt achtereenvolgens van regel 0 t/m regel 15 geladen. Dit betekent natuurlijk dat deze regels niet kris-kras door elkaar mogen staan, maar precies de voorschriften moeten volgen die ook gelden voor het direct intikken van de tekst. In het bijzonder betekent dit dat een opdracht pas uitgevoerd mag worden nadat die gedefinieerd is.

Merk op dat alles in dit blok direct ingetikt zou kunnen worden, maar dat er voordelen aan verbonden zijn om lange definities eerst op te maken tot blokken op schijf of cassette. In ieder programma dat je aan

het ontwikkelen bent, zitten fouten en het corrigeren van een blok, gevolgd door het opnieuw laden voor het testen, is eenvoudiger dan het in zijn geheel intikken van een programma. Een ander voordeel van het gebruik van blokken voor het ontwikkelen van programma's is dat goed opgemaakte en van commentaar voorziene blokken nauwelijks nadere documentatie vereisen en goed bruikbaar zijn voor andere programmeurs.

```
0  ( Voorbeelden uit hoofdstuk 5 van dit boek )
1
2  : kwadraten              ( druk het kwadraat van 0 t/m 9 af )
3    10 0 DO
4           I I * .
5         LOOP ;
6  kwadraten                ( test van kwadraten )
7
8  : *tabel                 ( druk een vermenigvuldigingstabel af )
9    CR 11 1 DO
10           11 1 DO
11                 J I * .
12               LOOP
13           CR
14           LOOP ;
15 *tabel                   ( test van *tabel )
```

De meeste systemen gebruiken meer dan een blokbuffer (meestal twee of drie); het systeem beslist automatisch welk blok voor een zekere LOAD of LIST zal worden gebruikt (veelal wordt de blokbuffer die het langst niet gebruikt is hiervoor genomen). Ter toelichting van wat dit betekent, veronderstellen we dat we in blok 100 een aantal colondefinities aan het ontwikkelen zijn en dat FORTH blokbuffer 1 (blokbuffer 2 is de andere voor een systeem met twee blokbuffers) toegewezen heeft voor blok 100. Als we dan voor naslag blok 95 willen afdrukken, zal FORTH daarvoor blokbuffer 2 gebruiken omdat die het langst niet gebruikt is. Na blok 95 bekeken te hebben, kunnen we de opmaak van blok 100 vervolgen met {100 LIST}; blok 100 hoeft dan niet opnieuw van schijf of cassette gelezen te worden omdat het nog in blokbuffer 1 zit. Op deze manier kunnen we het lezen van of het schrijven op schijf of cassette beperken terwijl we een bepaald blok aan het ontwikkelen zijn.

Als we in het voorgaande geval {96 LIST} ingetikt hadden (om blok 96 te bekijken) in plaats van {100 LIST} (om de opmaak van blok 100 te vervolgen), dan zou buffer 1 gebruikt worden voor blok 96, omdat buffer 2 het laatst gebruikt was. Maar daaraan voorafgaand zal FORTH automatisch de inhoud van buffer 1 teruggeschreven hebben naar blok 100, zodat geen informatie verloren gaat. De FORTH programmeur hoeft dus niet zelf blokken 'te redden'.

De basisopdracht voor het lezen van een blok van schijf of cassette is {BLOCK}, die dan ook gebruikt wordt om LIST en LOAD te definiëren. Het effect van deze opdracht is dat het genoemde blok gekopieerd wordt op de langst niet gebruikte blokbuffer (tenzij het blok daar al was), nadat zo nodig de inhoud van die blokbuffer in veiligheid is gebracht. Met de inhoud van de buffer gebeurt verder niets, maar het beginadres van de buffer wordt wel op de stapel gezet:

```
BLOCK                (n → adres)
```

en vervangt daar dus het bloknummer. Met behulp van dit adres kun je dan bepaalde gegevens bereiken en die gebruiken of veranderen. Als voorbeeld volgt nu de definitie van de opdracht {INDEX}, waarmee regel 0 van een aantal gespecificeerde blokken afgedrukt moet worden:

```
: INDEX
      1+ SWAP DO              ( lus voor blokken )
              I BLOCK         ( lees een blok )
              64 0 DO         ( lus voor regel 0 )
                    DUP C@ EMIT 1+
                 LOOP
              DROP CR         ( stapel bijwerken, nieuwe regel )
            LOOP ;
```

Het afdrukken van regel 0 van blokken 100 t/m 105 gaat vervolgens met:

```
100 105 INDEX
```

In de definitie van {INDEX} is de opdracht {EMIT} gebruikt, die in het volgende hoofdstuk uitvoerig besproken zal worden. Merk op hoe eenvoudig het is om een blok te lezen en er gegevens uit te halen.

De {BLOCK} opdracht kan net zo gemakkelijk gebruikt worden om numerieke gegevens in een geheugenblok te redden of ze eruit op te halen. Bijvoorbeeld:

```
CREATE data 40 ALLOT          ( maak een array met 40 plaatsen )

: red-data                    ( moet op blok 150 gebeuren )
          data                ( geeft adres van het array )
          150 BLOCK           ( haal het blok )
          40 MOVE             ( zet data in het blok )
          UPDATE ;            ( noteer bijgewerkt zijn )

: laad-data                   ( data uit blok 150 )
          150 BLOCK           ( lees het blok )
          data                ( geeft adres van array )
          40 MOVE ;           ( zet data in array )
```

In het blok zijn de numerieke gegevens opgeslagen in binaire vorm, hetgeen een efficiënte en compacte vastlegging is (we kunnen zo 512 enkele-nauwkeurigheid-getallen in een blok opslaan), maar het afdrukken van zo'n blok geeft iets onbegrijpelijks!

De opdracht {MOVE} wordt gebruikt om gegevens naar en van de blokbuffer over te brengen; de stapelnotatie voor {MOVE} is:

```
MOVE                    (adr1 adr2 n → )
```

waarbij n getallen (van 16 bits), in het geheugen beginnend op adres adr1, gekopieerd worden naar de geheugenplaatsen beginnend op adr2. De getallen worden daartoe één voor één gekopieerd, dus van adr1 naar adr2, van adr1 + 2 naar adr2 + 2, van adr1 + 4 naar adr2 + 4, enz. (let wel op dat voor een bepaalde waarde van n de adressen adr1 en adr2 zo gekozen worden dat de te kopiëren adressen niet overlappen met de adressen waarnaar gekopieerd moet worden).

Het eveneens nieuwe woord {UPDATE} heeft tot resultaat dat de nu in gebruik zijnde blokbuffer met daarin blok 150 gemerkt wordt met 'bijgewerkt'. Dit garandeert het 'redden' van blok 150 voordat deze blokbuffer voor een latere LIST, LOAD, BLOCK of SAVE-BUFFERS in gebruik wordt genomen. Wanneer een blok met editor woorden wordt veranderd, gebeurt dit merken automatisch; wordt een blok met een programma veranderd (zoals hierboven) dan moet het expliciet met {UPDATE} worden gemerkt!

Wanneer we een blok in gebruik willen nemen, is in plaats van {BLOCK} het woord {BUFFER} handiger omdat dit het effect heeft de langst niet meer gebruikte buffer toe te kennen aan het gespecificeerde blok (na de oude bufferinhoud gered te hebben wanneer het 'bijgewerkt' is zonder dit blok te lezen. De stapelnotatie is:

```
BUFFER                  (n → adres)
```

Deze opdracht wordt gebruikt in de volgende colon-definitie voor een opdracht om een blok helemaal 'schoon te vegen' (d.i. met spaties te vullen):

```
: schoon
        BUFFER              ( zoek een buffer )
        1024 32 FILL        ( vul die met spaties )
        UPDATE ;            ( noteer bijgewerkt zijn )
```

Om bijvoorbeeld blok 105 schoon te vegen, tikken we in:

```
105 schoon
```

want wanneer de bufferinhoud naar schijf of cassette wordt overgebracht (bijvoorbeeld door SAVE-BUFFERS) wordt de inhoud van het blok 105 met spaties gevuld. De verklaring voor {FILL} wordt in het volgende hoofdstuk gegeven.

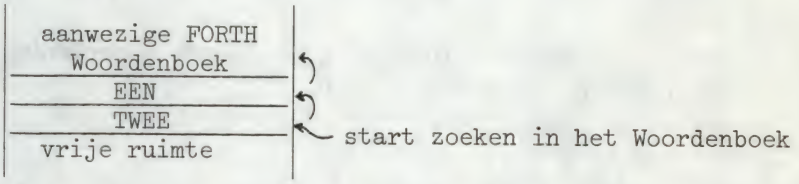
Ter afsluiting van deze paragraaf volgen nu nog drie woorden, nl. {SCR}, {BLK} en {EMPTY-BUFFERS}. Het eerste is een systeem-variabele (hoeft dus niet door de programmeur gedeclareerd te worden) met een waarde gelijk aan het bloknummer van het laatst afgedrukte blok

(handig voor nieuwe editor woorden). Het tweede is eveneens een systeem-variabele met een waarde gelijk aan het bloknummer van het blok dat momenteel als gevolg van een LOAD geïnterpreteerd wordt. De FORTH interpretator gebruikt deze variabele om na te gaan waar de invoer vandaan komt; is {BLK} nul dan is de invoer van het toetsenbord afkomstig en anders van een blokbuffer.

Door {EMPTY-BUFFERS} worden alle blokbuffers met 'leeg' gemerkt, zodat zelfs met een UPDATE geen van de buffers naar schijf of cassette wordt gekopieerd. Dit is handig wanneer je per ongeluk de inhoud van een buffer verknoeit (bijvoorbeeld tijdens correctie) en je niet wilt dat het oude blok hetzelfde lot ondergaat. Tik dan {EMPTY-BUFFERS} in, en herstel de buffer door het gewenste blok af te drukken.

6.4 Beheer van woordenboeken

Zoals al eerder opgemerkt, bestaat een compleet programma in het algemeen uit een aantal colon-definities, of anders gezegd een 'woordenschat' van nieuwe 'woorden'. Door het laden van een woordenschat worden de nieuwe woorden vertaald en 'gekoppeld' aan het bestaande Woordenboek. Figuur 6.2 laat deze koppeling zien voor de nieuwe woorden {EEN} en {TWEE}.

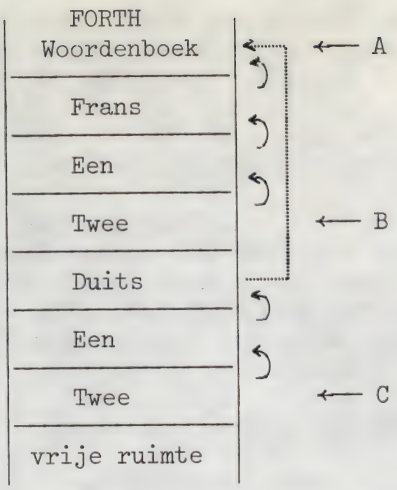


Figuur 6.2 Koppeling aan het Woordenboek

Het zoeken in het Woordenboek begint bij het laatst gedefinieerde woord en vandaar terug. Is het gezochte woord dus niet een van de toegevoegde woorden, dan wordt uiteindelijk in het standaard Woordenboek gezocht. Dank zij de koppeling mogen nieuwe en oude woorden dus door elkaar in een invoersliert voorkomen.

Iedere woordenschat kan op die manier geladen en gekoppeld worden en bij het zoeken van een woord wordt een woordenschat in omgekeerde volgorde van het laden ervan doorzocht. Hoewel dit bevredigend werkt, voorziet FORTH nog in een elegantere manier voor het koppelen van een woordenschat aan het standaard Woordenboek. Daarmee kan iedere woordenschat afzonderlijk bekeken worden. In het volgende voorbeeld kun je zien hoe de woorden {VOCABULARY} en {DEFINITIONS} een rol spelen in de koppeling van twee aparte woordenlijsten, genaamd {Frans} en {Duits}, aan het FORTH Woordenboek. In figuur 6.3 is een en ander schematisch weergegeven. De wijzers A, B en C geven daarin aan dat op drie plaatsen gestart kan worden met het zoeken in het Woordenboek.

```
 0 ( Test woordenboekbeheer )
 1 VOCABULARY Frans IMMEDIATE
 2 Frans DEFINITIONS
 3 : Een ." un " ;
 4 : Twee ." deux " ;
 5
 6 FORTH DEFINITIONS
 7 VOCABULARY Duits IMMEDIATE
 8 Duits DEFINITIONS
 9 : Een ." eins " ;
10 : Twee ." zwei " ;
```



Figuur 6.3 Koppeling van twee woordenlijsten

We kunnen direct in het FORTH Woordenboek (bij A en daarmee de twee andere woordenboeken overslaan) starten met

```
FORTH ok
Een Een ?
```

We kunnen in B starten en dus eerst de Franse woordenlijst doorzoeken en daarna het FORTH Woordenboek met:

```
Frans ok
Een un ok
```

of we kunnen in C starten en dus de Duitse woordenlijst doorzoeken en daarna meteen FORTH met:

```
Duits ok
Twee zwei ok
```

Zoals aangegeven mogen in verschillende woordenlijsten dezelfde namen voor verschillende definities voorkomen. Iedere versie van datzelfde woord is te bereiken door voorafgaand aan dat woord de naam van de desbetreffende woordenlijst te noemen.

In FORTH heet de woordenlijst waarin het zoeken moet beginnen, de 'context' woordenlijst; het zoeken begint op de plaats vastgelegd met de systeem-variabele {CONTEXT}. Bij uitvoering van een woord (als 'Frans' of 'Duits') dat gedefinieerd is met {VOCABULARY}, krijgt {CONTEXT} de waarde van de daarmee corresponderende wijzer (B of C). Het woord {FORTH} geeft aan {CONTEXT} de waarde corresponderend met het FORTH Woordenboek.

Let ook op het gebruik van het woord {IMMEDIATE} na de definities van 'Frans' en 'Duits' in regels 1 en 7 van het voorgaande voorbeeld. Dit garandeert dat de opdrachten 'Frans' en 'Duits' altijd uitgevoerd worden, ook in een colon-definitie zoals we dadelijk zullen zien (zie ook 9.5).

Een andere systeem-variabele, {CURRENT}, legt vast in welke woordenlijst nieuwe definities geplaatst moeten worden; het woord {DEFINITIONS} geeft {CURRENT} de waarde van {CONTEXT}. Daarom betekent:

```
FORTH DEFINITIONS
```

dat nieuwe definities gekoppeld zullen worden aan punt A in figuur 6.3 en deel zullen uitmaken van de FORTH woordenlijst. De nieuwe definitie mag nog steeds gebruik maken van andere woordenlijsten door in de definitie de context te veranderen. Bijvoorbeeld:

```
FORTH DEFINITIONS ok
: twee-vertaling
          Frans Twee ." = "
          Duits Twee ; ok
FORTH twee-vertaling  deux = zwei ok
```

De woorden 'Frans' en 'Duits' worden tijdens de vertaling uitgevoerd (zonder een tekst te leveren) en veranderen de context zo dat de eerste {Twee} gevonden wordt in de Franse woordenlijst en de tweede {Twee} in de Duitse.

6.5 Samenvatting

In dit hoofdstuk zijn de volgende nieuwe FORTH woorden ingevoerd:

Invoer van en uitvoer naar achtergrondgeheugen:

LIST (n →)
De inhoud van blok n wordt afgedrukt, SCR krijgt de waarde n.

LOAD (n →)
Lees de invoer van blok n en interpreteer het (bewaar de wijzers {>IN} en {BLK} naar de huidige invoer, zodat die weer opgenomen kan worden wanneer de interpretatie van het blok eindigt).

BLOCK (n → adres)
Wanneer blok n niet reeds in het geheugen is, wordt het uit het achtergrondgeheugen gekopieerd in de langst niet meer gebruikte blokbuffer (nadat de vorige inhoud van die blokbuffer gered is indien die inhoud gewijzigd is met een {UPDATE}). Het adres van de blokbuffer wordt voorts op de stapel gezet.

BUFFER (n → adres)
Reserveer de langst niet meer gebruikte blokbuffer voor blok n nadat de inhoud daarvan gered is indien die inhoud gewijzigd is met een {UPDATE}. Blok n uit het achtergrondgeheugen wordt niet in de blokbuffer gekopieerd; het adres van die buffer wordt wel op de stapel gezet.

SCR (→ adres)
Deze systeem-variabele bevat het nummer van het blok dat het laatst afgedrukt is.

UPDATE (→)
Geef de laatst gebruikte blokbuffer het kenmerk 'gewijzigd', zodat de inhoud ervan gered wordt in het achtergrondgeheugen als deze buffer (door een LIST, LOAD of BLOCK) voor een ander blok nodig is of een SAVE-BUFFERS wordt uitgevoerd.

SAVE-BUFFERS (→)
Red alle buffers met het kenmerk 'gewijzigd'.

EMPTY-BUFFERS (→)
Geef alle blokbuffers het kenmerk 'leeg' zonder die blokbuffers vooraf te redden, zelfs als ze het kenmerk 'gewijzigd' hebben.

Woordenboekbeheer

VOCABULARY (→)
Wanneer gebruikt in VOCABULARY <naam> wordt in de CURRENT woordenlijst een nieuwe woordenlijst, geheten <naam>, gemaakt. Bij latere uitvoering van <naam> zal dit de CONTEXT woordenlijst worden voor het zoeken van een woord. Met de constructie <naam> DEFINITIONS wordt <naam> de CURRENT woordenlijst voor nieuwe definities. Alle woordenlijsten zijn gekoppeld met FORTH.

CURRENT (→ adres)
Een systeem-variabele die aangeeft aan welke woordenlijst nieuwe woorden zullen worden toegevoegd.

CONTEXT (→ adres)
Een systeem-variabele die aangeeft in welke woordenlijst het opzoeken van een woord zal beginnen voor de interpretatie van de invoer.

FORTH (→)
Deze standaard woordenlijst is gewoonlijk zowel de CONTEXT als de CURRENT woordenlijst; dit is te veranderen met VOCABULARY en DEFINITIONS. Door uitvoering van FORTH wordt dit weer de CONTEXT woordenlijst.

DEFINITIONS (→)
Maakt CURRENT gelijk aan CONTEXT, zodat volgende definities gekoppeld worden aan de woordenlijst die eerder bekend stond als CONTEXT woordenlijst.

Diversen

MOVE (adr1 adr2 n →)
Kopieert n getallen (van 16 bits), beginnend op adres adr1 in het geheugen, naar geheugenplaatsen, beginnend op adr2. Is n niet groter dan 0, dan gebeurt er niets.

PAD (→ adres)
Zet het beginadres van een 'kladregel' (met tenminste 64 karakters) op de stapel.

BLK (→ adres)
Systeem-variabele met het nummer van het nu geïnterpreteerde blok (BLK = 0 geeft het toetsenbord aan).

7 INVOER EN UITVOER VAN GETALLEN EN STRINGS

We kennen al de {.} en {."} bewerkingen voor de uitvoer van resp. getallen en tekst (karakter-slierten) en voor de invoer van getallen hebben we er gebruikt van gemaakt dat getallen in de invoer automatisch op de stapel terecht komen. FORTH voorziet echter in een uitgebreide aanvullende verzameling van invoer- en uitvoer-operaties, die gebruikt worden voor het schrijven van nieuwe in- en uitvoer opdrachten (woorden) voor elke denkbare toepassing.

Om de tekst van dit hoofdstuk (en volgende) niet onnodig ingewikkeld te maken, worden voortaan veel voorbeelden zonder uitvoerige (stapel-) analyse gegeven. Met de methode uit paragraaf 3.6 kan de lezer die voorbeelden echter zelf in detail bestuderen.

7.1 De basis: invoer en uitvoer van karakters

De eenvoudigste uitvoerbewerking is de opdracht {EMIT}, die een karakter zal afdrukken waarvan de ASCII-waarde op de stapel staat (zie de verklarende lijst van FORTH woorden achter in dit boek voor een beschrijving van ASCII). Bijvoorbeeld:

```
65 EMIT Aok
```

zal het enkele karakter "A" afdrukken, omdat 65 de (decimale) ASCII-waarde is voor het karakter "A". Voor het afdrukken van een 'string' van karakters gebruiken we een aantal opeenvolgende {EMIT}'s:

```
89 EMIT 69 EMIT 83 EMIT YESok
```

Om niet steeds in een tabel de ASCII-waarde van een karakter op te moeten zoeken, kunnen we de omgekeerde {KEY} bewerking gebruiken; deze wacht totdat een toets is ingedrukt en zet dan de waarde daarvan op de stapel. Bijvoorbeeld:

```
KEY ok                  ( druk na 'return' de toets 'Y' in )
. 89 ok
```

Voor het afdrukken van strings is {."} echter veel gemakkelijker:

```
." YES" YESok
```

Hiermee kun je echter niet de ASCII-karakters afdrukken, waarmee je o.a. de opmaak van de afgedrukte tekst verzorgt, zodat je voor die karakters wel {EMIT} moet gebruiken. Dit gebeurt in de volgende standaard FORTH-woorden:

```
: CR  13 EMIT  10 EMIT ;            ( cursor terug, nieuwe regel )
: SPACE 32 EMIT ;                   ( druk 1 spatie af )
: SPACES 0 DO SPACE LOOP ;          ( druk n spaties af )
```

Dit illustreert dat veel standaard FORTH-woorden zelf in FORTH gedefinieerd zijn! Zo nodig kunnen andere speciale afdrukbewerkingen zelf gedefinieerd worden, bijvoorbeeld:

```
: TAB  9 EMIT ; ok          ( cursor naar volgende tabulatorstand )
: CLRS 12 EMIT ; ok         ( maak scherm schoon )
CLRS TAB ." Nieuwe pagina" CR

            Nieuwe pagina
ok
```

Laten we voor het laatste voorbeeld van het gebruik van zowel {EMIT} als {KEY}, aannemen dat we met het toetsenbord een string van bepaalde lengte willen inlezen en later weer afdrukken. De beste manier om in het geheugen ruimte voor de string te reserveren is al in paragraaf 3.5 gegeven:

```
CREATE string 6 ALLOT ok     ( ruimte voor 6 bytes )
```

Het woord {string} zal het adres van de gereserveerde ruimte op de stapel zetten en kan als volgt in nieuwe definities worden gebruikt:

```
: GETSTR CR ." ?"               ( druk ? af )
         string                 ( adres op stapel )
         6 0 DO                 ( lus voor 6 karakters )
               KEY              ( lees toetsaanslag in )
               DUP EMIT         ( druk aangeslagen toets af )
               OVER C!          ( onthoud aangeslagen toets )
               1+
             LOOP
         DROP ; ok              ( stapel bijwerken )

: PRINTSTR
         string                 ( adres op stapel )
         6 0 DO                 ( lus voor 6 karakters )
               DUP C@ EMIT      ( pak karakter en druk het af )
               1+
             LOOP
         DROP ; ok              ( stapel bijwerken )

GETSTR                          ( test van GETSTR )
?ABCDEFok
PRINTSTR ABCDEFok               ( test van PRINTSTR )
```

Twee beperkingen van {GETSTR} zijn dat altijd 6 karakters ingetikt mòeten worden (en afsluiting met de 'return'-toets niet mogelijk is) en dat de 'backspace' ('terug') toets niet gebruikt kan worden voor het

corrigeren van tikfouten. Met een slimmere definitie van {GETSTR} zou je deze beperkingen kunnen opheffen, maar omdat FORTH al een aantal krachtige invoerbewerkingen voor strings heeft is het verstandiger om die te gebruiken.

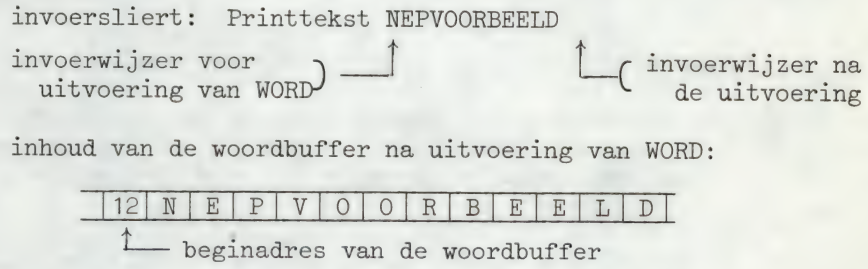
7.2 Invoer en uitvoer van strings I

De FORTH programmeur heeft in het algemeen twee mogelijkheden voor de invoer van strings in programma's. De eerste is de string uit de oorspronkelijke invoersliert te halen (hiervoor bestaat geen equivalent in BASIC). De tweede manier bestaat uit het stoppen van het programma, gevolgd door het intikken van de string op het toetsenbord (zoals in {GETSTR} of met de 'INPUT'-opdracht van BASIC). De eerste manier, die in deze paragraaf wordt beschreven, maakt gebruik van de belangrijke bewerking {WORD}. De tweede manier wordt in de volgende paragraaf besproken.

Het volgende voorbeeld laat het gebruik van {WORD} zien. Deze opdracht is alleen in een colon-definitie toegestaan (zoals later zal worden uitgelegd):

```
: Printtekst 32 WORD COUNT TYPE ; ok
Printtekst NEPVOORBEELD NEPVOORBEELDok
```

{Printtekst} heeft het weinig zinvolle effect dat het daarna ingetikte woord weer afgedrukt wordt, maar deze opdracht illustreert enkele interessante nieuwe bewerkingen. In de eerste plaats heeft {WORD} bij uitvoering het resultaat dat karakters uit de invoersliert gekopieerd worden in een 'woordbuffer' tot een gekozen 'slotkarakter' ontmoet wordt. Bovendien wordt het aantal gekopieerde karakters vooraan in de woordbuffer gezet. In figuur 7.1 is dit schematisch weergegeven.



Figuur 7.1 De werking van {WORD}

Het gekozen slotkarakter moet vooraf op de stapel gezet worden en achteraf krijg je hiervoor in de plaats het beginadres van de woordbuffer.

```
WORD                (kar → adres)
```

Het resultaat van {32 WORD} is dus dat gekopieerd wordt tot de volgende spatie (want 32 is de ASCII waarde van een spatie).

Het resultaat van het woord {COUNT} uit het voorbeeld is dat het adres op de stapel met 1 wordt verhoogd en dat het aantal in de woordbuffer gekopieerde karakters boven op de stapel wordt gezet:

```
COUNT                (adres → adres+1 n)
```

De stapel bevat nu de gegevens voor {TYPE}, waarmee de string wordt afgedrukt waarvan het beginadres en het aantal karakters op de stapel staan:

```
TYPE                 (adres n →  )
```

De opdracht {WORD} mag niet buiten een colon-definitie worden gebruikt omdat FORTH voor het verwerken van een invoersliert deze opdracht zelf benut en dientengevolge ook de woordbuffer. Intikken van:

```
32 WORD nepwoord COUNT TYPE
```

zal dan niet het gewenste resultaat hebben omdat tegen de tijd dat {TYPE} wordt uitgevoerd, de woordbuffer niet "nepwoord" bevat (maar in werkelijkheid "TYPE"!).

Opmerking. Een aantal FORTH-systemen reageert anders op {WORD}; deze systemen zetten het adres van de woordbuffer niet op de stapel. Voor die systemen moet je mogelijk de opdracht {WORD} vervangen door {WORD HERE}, maar kijk dit eerst na in je documentatie!

Nu we weten hoe {WORD} werkt, kunnen we een (slimmere) opdracht, {PUTSTR}, ontwerpen die de daarop volgende string in de invoersliert plaatst in een 'stringbuffer', die we zelf definiëren:

```
CREATE string 40 ALLOT ok          ( zal groot genoeg zijn )
: PUTSTR
     string 40 32 FILL             ( veeg stringbuffer schoon )
     32 WORD COUNT                 ( string in woordbuffer )
     string SWAP CMOVE ; ok        ( kopieer naar stringbuffer )
```

Hierbij zijn twee nieuwe woorden geïntroduceerd. Hiervan heeft {FILL} de handige eigenschap een aantal opeenvolgende byte-posities in het geheugen te vullen met eenzelfde karakter. De stapelbeschrijving is:

```
FILL                 (adres n kar →  )
```

waarin {adres} het begin aangeeft van n opeenvolgende byte-posities die met {kar} worden gevuld. De opdrachtenreeks {string 40 32 FILL} zorgt er dus voor dat 40 bytes, beginnend bij het door {string} opgeleverde adres, met spaties (ASCII code 32) gevuld worden.

De opdracht {CMOVE} met de stapelbeschrijving:

```
CMOVE                (adr1 adr2 n →  )
```

verplaatst een blok van n karakters, beginnend op adr1 naar adr2. In het voorgaande voorbeeld wordt de inhoud van de woordbuffer dus overgebracht naar de zelf gedefinieerde stringbuffer. De inhoud daar-

van kan niet verknoeid worden door het FORTH-systeem.

Met een nieuwe definitie voor {PRINTSTR} kunnen we {PUTSTR} testen:

```
: PRINTSTR string 40 -TRAILING TYPE ; ok

PUTSTR teststring ok
PRINTSTR teststringok
```

In {PRINTSTR} is een andere handige stringbewerking {-TRAILING} gebruikt; deze vermindert het aantal karakters op de top van de stapel met het aantal spaties achteraan de opgeslagen string (waarvan het adres ook op de stapel staat). Daarom zal {TYPE} alleen de niet-spaties afdrukken. Zonder het woord {-TRAILING} zou {PRINTSTR} altijd alle 40 karakters in de stringbuffer afdrukken (dus inclusief de spaties achteraan).

Tot slot van deze paragraaf moeten nog twee woorden genoemd worden die in samenhang met {WORD} van nut kunnen zijn. Het woord {HERE} zet het adres van de eerste vrije plaats in het Woordenboek op de stapel (in veel systemen is dat hetzelfde adres als {WORD} zou opleveren omdat de woordbuffer opschuift als het Woordenboek groeit). Voorts het woord {>IN }, dat een systeem-variabele is waarvan de waarde gelijk is aan de positie van de invoerwijzer in figuur 7.1. Terwijl {WORD} de invoersliert verwerkt, wordt tegelijkertijd de waarde van {>IN } groter.

7.3 Invoer en uitvoer van strings II

De tweede manier voor het invoeren van strings maakt gebruik van het woord {EXPECT} of van het woord {QUERY}. Beiden stoppen de verwerking van het programma totdat op het toetsenbord een invoersliert is ingetikt; het verschil zit alleen in de plaats waar die sliert wordt opgeborgen.

De opdracht {EXPECT} met de stapelbeschrijving:

```
EXPECT                (adres n →  )
```

bergt de op het toetsenbord ingetikte karakters op in het geheugen, beginnend bij 'adres', totdat hetzij n karakters ingetikt zijn, hetzij de 'return'-toets is aangeslagen. We kunnen hiermee bijvoorbeeld een heel nuttige opdracht, {INSTR}, definiëren waarmee karakters van het toetsenbord in onze stringbuffer geplaatst kunnen worden:

```
: INSTR string 40 32 FILL          ( veeg stringbuffer schoon )
       CR ." ?"                    ( druk ? af )
       string 40 EXPECT ; ok       ( verwacht hoogstens 40 kar.)

INSTR
?Dit is een voorbeeld ok

PRINTSTR Dit is een voorbeeldok
```

De bewerking {QUERY} doet vrijwel hetzelfde als {EXPECT}; alleen worden nu tot 80 karakters toe geplaatst in de invoerbuffer voor een werkstation. Het woord {QUERY} wordt eveneens door het FORTH-systeem gebruikt wanneer je gewoon via het toetsenbord iets invoert. In veel opzichten is dit woord handiger voor de FORTH programmeur dan het woord {EXPECT} omdat het samen met {WORD} gebruikt kan worden om hetgeen ingetikt is te ontleden in de afzonderlijke woorden.

Laten we bijvoorbeeld aannemen dat we een programma aan het schrijven zijn met een 'vraag en antwoord' dialoog tussen de computer en de gebruiker (zoals in sommige computer spelletjes), waarbij de gebruiker op een regel drie antwoorden, gescheiden door komma's, moet intikken. Met een slimme combinatie van {QUERY} en {WORD} kunnen we met de volgende colon-definitie de drie antwoorden in aparte stringbuffers zetten:

```
CREATE antw1 10 ALLOT ok            ( definieer 3 stringbuffers )
CREATE antw2 10 ALLOT ok
CREATE antw3 10 ALLOT ok

: zetantwoorden
      antw1 10 32 FILL              ( veeg die buffers schoon )
      antw2 10 32 FILL
      antw3 10 32 FILL
      CR ." ?"                      ( nieuwe regel en vraagteken )
      QUERY                         ( ontvang antwoord )
      44 WORD                       ( pak eerste antwoord )
      COUNT antw1 SWAP CMOVE        ( zet het in de eerste buffer )
      44 WORD                       ( pak tweede antwoord )
      COUNT antw2 SWAP CMOVE        ( zet het in de tweede buffer )
      1 WORD                        ( pak derde antwoord )
      COUNT antw3 SWAP CMOVE ; ok   ( zet het in de derde buffer )

zetantwoorden
?Een, Twee, Drie ok

antw1 10 TYPE Een       ok          ( druk antwoorden af )
antw2 10 TYPE Twee      ok
antw3 10 TYPE Drie      ok
```

Deze definitie is in de praktijk natuurlijk een onderdeel van andere definities, die samen het programma vormen. Het 'eindkarakter' voor de eerste twee antwoorden is 44, de ASCII-waarde voor een komma, en voor het derde antwoord een 1 die maakt dat alle overblijvende karakters tot het einde van de regel gekopieerd worden. Merk op dat we de stringbuffers de korte namen 'antw1', enz., hebben gegeven in plaats van 'antwoord1', enz. Dit gebeurde omdat sommige FORTH-systemen alleen de eerste vijf of zes karakters opbergen in de Woordenlijst, zodat dan 'antwoord1', 'antwoord2' en 'antwoord3' niet uit elkaar zijn te houden. Voor systemen die slechts de eerste drie of vier karakters onderscheiden, moet je namen als '1antw', enz. invoeren.

Nog twee eigenschappen van {QUERY} verdienen de aandacht. De eerste is dat dit woord de invoerbuffer van het werkstation gebruikt en dat dus alles wat eerder in de buffer stond, inclusief FORTH invoer, verdwijnt. Daarom zal in:

```
zetantwoorden ." Dit gaat niet"
```

{zetantwoorden} correct worden uitgevoerd, maar wat daarop volgt wordt niet uitgevoerd. De tweede is dat wanneer de systeemvariabele {BLK} (uit hoofdstuk 6) niet nul is, de opdracht {QUERY} zal trachten invoer van schijf of cassette te halen. Dit is handig als we tijdens uitvoering van een programma stringinvoer uit een blok op schijf nodig hebben.

7.4 Radix van getallen

Een nog niet gebruikte mogelijkheid bij invoer of uitvoer van getallen is dat de radix ('grondtal') anders dan de gebruikelijke 10 (het tientallig stelsel) mag zijn. De waarde van de radix is vastgelegd in de systeem-variabele {BASE}, zodat radix-conversie heel eenvoudig gaat in FORTH omdat het alleen de definitie van enkele woorden vergt om de waarde van {BASE} te veranderen. Bijvoorbeeld:

```
DECIMAL ok
: HEX 16 BASE ! ; ok
: BIN  2 BASE ! ; ok
```

De net gedefinieerde woorden {HEX} en {BIN} zullen bij uitvoering bewerkstelligen dat de radix voor invoer of uitvoer van getallen dan 16 ('hexadecimaal'), resp. 2 ('binair') is. Als voorzorg is voor de definitie van deze woorden {DECIMAL} ingetikt om er zeker van te zijn dat de getallen 16 en 2 als decimale getallen worden opgevat tijdens de vertaling. Dit woord {DECIMAL} is een standaard-woord, dat {BASE} de waarde 10 geeft, de 'normale' waarde voor decimale invoer en uitvoer.

Met {HEX} en {BIN} is heel eenvoudig radix-conversie te doen, zoals:

```
DECIMAL                    ( decimale invoer )
16 HEX . 10 ok             ( decimaal 16 = hexadecimaal 10 )
3 FF DECIMAL . 1023 ok     ( hexadecimaal 3FF = decimaal 1023 )
9 BIN . 1001 ok            ( decimaal 9 = binair 1001 )
10101 DECIMAL . 21 ok      ( binair 10101 = decimaal 21 )
```

Voor de conversie van negatieve getallen is {U.} nuttiger dan {.}:

```
-1 HEX U. FFFF ok          ( decimaal -1 = hexadecimaal FFFF )
FFFF DECIMAL U. 65535 ok
```

Het laatste is correct, want het getal met teken -1 en het getal zonder teken 65535 worden in de machine door dezelfde 16 bits voorgesteld!

Wanneer je verschillende radix-systemen door elkaar gebruikt, kun je wel eens vergeten met welk systeem je bezig bent. Intikken van

```
BASE @ . 10 ok
```

helpt je dan niet omdat je met iedere radix 10 krijgt! Is bijvoorbeeld de radix (en dus 'BASE') 16, dan wordt dat hexadecimaal als 10 afgedrukt. Dit probleem is op te lossen met een speciale definitie:

```
: ?BASE
      BASE @            ( huidige radix )
      DUP               ( dupliceer deze )
      DECIMAL .         ( druk deze decimaal af )
      BASE ! ; ok       ( herstel de oude radix )
```

die altijd gebruikt kan worden:

```
HEX ?BASE 16 ok
BIN ?BASE 2 ok
DECIMAL ?BASE 10 ok
```

Een ongewone, maar handige toepassing van het gebruik van een willekeurige radix in FORTH is, dat korte strings opgevat kunnen worden als getallen met radix 36. Iedere string van "A" tot "ZZZ" kan met radix 36 worden voorgesteld door een getal van 16 bits zonder teken. Het vergelijken van strings met ten hoogste drie karakters is dan met gewone rekenbewerkingen uit te voeren (door te rekenen met dubbele nauwkeurigheid (zie volgende hoofdstuk) kunnen we zelfs strings met ten hoogste zes karakters eenvoudig met elkaar vergelijken). In het volgende voorbeeld wordt bijvoorbeeld een uit korte strings bestaand array gedefinieerd alsof het een array met getallen is.

```
DECIMAL ok
: 36basis 36 BASE ! ; ok

36basis ok
CREATE dagen ZND , MND , DSD , WSD , DRD , VRD , ZRD , ok

DECIMAL ok
: .dag ( druk weekdag, genummerd met 0 - 6, verkort af )
     BASE @ SWAP          ( bewaar huidige radix )
     2 * dagen + @        ( zet dag op de stapel )
     36basis U.           ( druk die af )
     BASE ! ; ok          ( herstel oude radix )

3 .dag  WSD ok
6 .dag  ZRD ok
```

7.5 Een alternatief voor de invoer van getallen

In de meeste FORTH programma's worden getallen via de stapel ingevoerd. De voor een programma benodigde gegevens worden op de stapel gezet voordat het programma verwerkt wordt (door de naam ervan te geven), en het programma haalt de gegevens weer tijdens de verwerking van de stapel af. In sommige gevallen is echter een meer gebruikelijke invoer van getallen nodig, waarbij een programma stopt totdat een benodigd getal op het toetsenbord ingetikt is (zoals bij de 'INPUT'-opdracht in BASIC, wanneer die voor invoer van getallen wordt benut).

Hoewel FORTH in zijn standaard Woordenlijst geen numerieke 'INPUT'-opdracht heeft, kunnen we er zelf een definiëren met gebruikmaking van de gebufferde invoermethode van paragraaf 7.3 en het woord {CONVERT}. Dit woord zet een ASCII-string uit het geheugen om in een 'dubbele nauwkeurigheid' getal op de stapel. Alles wat we te doen hebben, is het in het geheugen plaatsen met {EXPECT} of {QUERY} van een invoersliert afkomstig uit het toetsenbord, gevolgd door {CONVERT}:

```
: INPUT
        0 0             ( dubbele nul op de stapel )
        CR ." ?"        ( druk een vraagteken af )
        QUERY           ( lees een string in )
        1 WORD          ( kopieer die in de woordbuffer )
        CONVERT         ( zet die om in een getal )
        DROP DROP ; ok  ( stapel bijwerken )
```

Hierbij doet 'CONVERT' het volgende met de stapel:

```
CONVERT                 (d1 adres1 → d2 adres2)
```

De string met startadres adres1 in de woordbuffer wordt 'omgezet' in een dubbele nauwkeurigheid getal, hierbij wordt d1 opgeteld en het resultaat d2 komt op de stapel. Verder is adres2 het adres van het eerste stringkarakter dat niet meer omgezet kan worden. De dubbele nul in {INPUT} representeert d1 en {DROP DROP} verwijdert adres2 en het minst belangrijke deel van d2, zodat een enkele nauwkeurigheid resultaat op de stapel overblijft (zie het volgende hoofdstuk voor details).

Opmerking. Sommige FORTH-systemen hebben in plaats van {CONVERT} het woord {NUMBER} met het stapel-effect (adres → d). Met dit woord moet in {INPUT} de laatste {DROP} en in het begin 0 0 weggelaten worden.

Voor het gebruik van {INPUT} moet het in een programma worden opgemen op die plaatsen waar met behulp van het toetsenbord getalinvoer nodig is. Er wordt dan gewacht totdat een getal is ingetikt (gevolgd door het aanslaan van de return-toets), waarna dat getal op de stapel staat:

```
INPUT                   ( → n)      Voer n in met het toetsenbord
```

Enkele voorbeelden voor het gebruik van {INPUT} zijn:

```
INPUT
?1234 ok                    ( tik in "1234" en return )
. 1234 ok                   ( druk het af met . )

36basis ok
INPUT
?Ja ok                      ( tik in "Ja" en return )
U. Ja ok                    ( druk het af met U. )

DECIMAL ok                  ( weer decimale radix )
```

Het laatste voorbeeld laat zien dat {CONVERT} ook {BASE} gebruikt om de radix vast te stellen van de te converteren string. Door {BASE} te veranderen kunnen we ook andere dan decimale getallen met {INPUT} inlezen.

7.6 Samenvatting

In dit hoofdstuk zijn de volgende nieuwe woorden ingevoerd:

Invoer en uitvoer van karakters

EMIT (kar →)
Druk het karakter af waarvan de ASCII-waarde op de stapel staat.

SPACE (→)
Druk een spatie af.

SPACES (n →)
Druk n spaties af als n groter dan 0 is en doe anders niets.

TYPE (adres n →)
Druk n karakters af, in het geheugen beginnend bij adres; doe niets als n niet groter dan 0 is.

COUNT (adres → adres+1 n)
Zet het aantal karakters van de string, aangewezen door adres, op de stapel. Verhoog adres met 1 (zodat daarmee nu het eerste karakter van de string wordt aangewezen); wat nu op de stapel staat zou met TYPE afgedrukt kunnen worden. $0 \leq n \leq 255$.

-TRAILING (adres n1 → adres n2)
Verminder het getal op de stapel met het aantal spaties achteraan de door adres aangewezen string waarop geen ander karakter meer volgt.

EXPECT (adres n →)
Breng karakters over van het toetsenbord naar het geheugen, beginnend bij adres, totdat òf n karakters zijn overgebracht of de return-toets is aangeslagen. Voeg daar nog een of twee uit nullen bestaande bytes aan toe. Doe niets als n niet groter dan 0 is.

QUERY (→)
Breng karakters over van het toetsenbord naar de invoerbuffer van het werkstation totdat òf 80 karakters zijn overgebracht òf de return-toets is aangeslagen. Deze string kan verder bewerkt worden met {WORD} wanneer de systeem-variabelen {>IN} en {BLK} beiden 0 zijn.

WORD (kar → adres)
Kopieer karakters van de invoerbuffer van het werkstation naar de woordbuffer, beginnend met het eerste karakter dat niet kar is en eindigend met het laatste karakter voorafgaand aan de volgende kar of met het laatste karakter in de invoerbuffer. Het aantal gekopieerde karakters wordt vooraan de gekopieerde string geplaatst en het adres van dat aantal wordt op de stapel gezet. Dit aantal is 0 als de invoerbuffer leeg is.

Invoer en uitvoer van getallen

BASE (→ adres)
Systeem-variabele die de huidige radix voor de invoer of uitvoer van getallen aangeeft.

DECIMAL (→)
Geeft {BASE} de waarde 10 (decimale getallen).

CONVERT (d1 adres1 → d2 adres2)
Zet de string, beginnend op adres1, om in een dubbele nauwkeurigheid getal, telt er d1 bij op en zet het resultaat d2 op de stapel. Voorts geeft adres2 het eerste stringkarakter aan dat, gezien de huidige radix, niet meer omgezet kon worden.

Diversen

CMOVE (adres1 adres2 n →)
Verplaats n bytes, beginnend op adres1, naar de geheugenposities, beginnend bij adres2. Doe niets als n niet groter dan 0 is.

FILL (adres n kar →)
Plaats het karakter kar op n opeenvolgende byte-posities in het geheugen, beginnend op adres.

HERE (→ adres)
Zet het adres van de eerste vrije plaats in het Woordenboek op de stapel.

>IN (→ adres)
Systeem-variabele die een positie in de woordbuffer aangeeft.

8 DUBBELE NAUWKEURIGHEID EN NOG MEER

In dit hoofdstuk worden enkele interessante bijzonderheden besproken over rekenen met dubbele nauwkeurigheid en het afdrukken van getallen met een zekere indeling. Voorts wordt aangegeven hoe je een reeks opdrachten voor het rekenen met tientallige breuken zou kunnen ontwerpen.

Voor de opdrachten, voorkomend in de uitgebreide FORTH-79 Woordenlijst, zijn ook de equivalente colon-definities gegeven ten behoeve van de bezitters van FORTH-systemen zonder deze uitbreiding.

8.1 Dubbele nauwkeurigheid getallen

In hoofdstuk 1 werd al vermeld dat het in FORTH ook mogelijk is om met dubbele nauwkeurigheid te rekenen. Dubbele nauwkeurigheid getallen (van 32 bits) met teken moeten liggen in het bereik:

```
-2 147 483 648 t/m 2 147 483 647
```

en die zonder teken in het bereik

```
0 t/m 4 294 967 295
```

(voor de duidelijkheid is tussen groepjes van drie cijfers een spatie gezet, maar die mogen in FORTH niet worden gebruikt!). Door ergens in een getal een decimale punt te zetten ziet FORTH het getal als een getal met dubbele nauwkeurigheid (sommige systemen gebruiken hiervoor een komma of dubbelepunt). Intikken van bijvoorbeeld:

```
DECIMAL ok
1000000. ok
```

heeft het resultaat dat het dubbele nauwkeurigheid getal 'een miljoen' op de stapel wordt gezet. Om het er weer af te halen en af te drukken, moet de opdracht {D} worden gebruikt:

```
D. 1000000 ok
```

(zie paragraaf 8.4 voor een andere methode als je systeem geen {D .} kent). De decimaal-punt, bij de invoer nodig voor het herkennen van een dubbele nauwkeurigheid getal, wordt dus niet afgedrukt (later in dit hoofdstuk zullen we zien hoe we een getal met een zekere indeling kunnen afdrukken).

Merk verder op dat:

```
-0.010 D. -10 ok
-10.   D. -10 ok
```

want beide invoerslierten worden gelezen als het dubbele nauwkeurigheid getal 'min tien'. In de meeste FORTH-79 standaard systemen wordt echter het aantal cijfers achter de decimaal punt onthouden met de variabele {DPL}, die voor de eerste invoersliert dus de waarde 3 krijgt en voor de tweede invoersliert de waarde 0. Later zullen we zien hoe deze extra informatie gebruikt kan worden om het werken met decimale breuken te programmeren.

Een dubbele-nauwkeurigheid-getal vergt twee opeenvolgende posities op de stapel met bovenop de eerste 16 bits van het getal. Als we een klein getal, dat dubbel nauwkeurig is, afdrukken met twee afdrukopdrachten voor enkele-nauwkeurigheid-getallen (dit is de {.} opdracht) dan krijgen we dit:

```
100. ok
. 0 ok          ( druk de eerste helft af )
. 100 ok        ( druk de tweede helft af )
```

Doen we dit voor een groot dubbel-nauwkeurig-getal, dan krijgen we op het eerste gezicht onbegrijpelijke resultaten (toch is: $15 * 2^{15} + 16960 = 1000000$!):

```
1000000. ok
. 15 ok
. 16960 ok
```

In de FORTH-79 norm komen twee dubbele-nauwkeurigheid-rekenbewerkingen en één vergelijkingsbewerking voor:

D+	(d1 d2 → dsom)	Dubbele nauwkeurigheid optelling met dubbele nauwkeurigheid resultaat
DNEGATE	(d → -d)	Tekenomkering in dubbele nauwkeurigheid
D<	(d1 d2 → b)	b is alleen true als d1 kleiner is dan d2

In de stapelbeschrijvingen stellen d, d1, enz. dubbele nauwkeurigheid getallen voor, die ieder twee stapelposities vergen. Om bijvoorbeeld twee van die getallen bij elkaar op te tellen, tikken we:

```
1000000. 2000000. D+ D. 3000000 ok
```

en voor een dubbele nauwkeurigheid aftrekking gebruiken we de opdracht {DNEGATE} in de definitie:

```
: D- DNEGATE D+ ; ok
2000000. 1. D- D. 999999 ok
```

Op soortgelijke manier kunnen we ook andere dubbele nauwkeurigheid bewerkingen definiëren als we die nodig hebben. Bijvoorbeeld:

```
: 2DUP OVER OVER ;                    ( dupliceer d-getal )
: 2DROP DROP DROP ;                   ( haal d-getal van stapel )
: D0< SWAP DROP 0< ;                  ( test voor negatief d-getal )
: D0= OR 0= ;                         ( test voor d-getal = 0 )
: D= D- D0= ;                         ( test voor gelijke d-getallen )
: DABS DUP 0< IF DNEGATE THEN ;       ( maak d-getal positief )
```

8.2 'Gemengde' nauwkeurigheid

In de FORTH-79 norm komen vier 'gemengde' nauwkeurigheid rekenbewerkingen voor, die betrekking hebben op combinaties van enkele en dubbele nauwkeurigheid getallen:

*/	(n1 n2 n3 → quotient)	Bereken het dubbele nauwkeurigheid product van twee enkele nauwkeurigheid getallen n1 en n2 en deel dat dan door een enkele nauwkeurigheid getal n3 om een enkele nauwkeurigheid quotient te krijgen.
*/MOD	(n1 n2 n3 → rest quotient)	Als het voorgaande, maar bovendien een enkele nauwkeurigheid rest.
U*	(un1 un2 → udproduct)	Vermenigvuldig twee enkele nauwkeurigheid getallen zonder teken met elkaar voor een dubbele nauwkeurigheid product zonder teken.
U/MOD	(ud un → urest uquot)	Deel het dubbele nauwkeurigheid getal ud door het enkele nauwkeurigheid getal un en zet de enkele nauwkeurigheid rest en quotient op de stapel. Alle getallen zonder teken!

De eerste twee bewerkingen zijn vooral opgenomen ter voorkoming van 'overloop'-problemen in berekeningen waarin een vermenigvuldiging gevolgd wordt door een deling. Ter toelichting nemen we aan dat we het 6/7-de deel van een aantal getallen moeten berekenen. Met:

```
: Breuk 6 * 7 / ; ok
100 Breuk . 85 ok
10000 Breuk . -790 ok          ( fout!! )
```

krijgen we in het eerste geval een correct (afgerond) resultaat, maar in het tweede geval een (blijkens het min-teken) kennelijk verkeerd antwoord. Dit komt omdat het product 10000 * 6 (= 60000) groter is dan het grootste toegestane enkele-nauwkeurigheid-getal 32767 (= 2^{15} - 1). Een dubbele-nauwkeurigheid-product van 10000 * 6 is wel toegestaan, zodat

```
: BREUK 6 7 */ ; ok
10000 BREUK . 8571 ok
```

deze nieuwe definitie voor alle enkele-nauwkeurigheid-getallen het goede resultaat aflevert.

De eerste bewerking maakt ook het werken met decimale breuken mogelijk. We kunnen bijvoorbeeld de bewerking 'maal π' definiëren met:

```
: *pi 31416 10000 */ ; ok
```

en krijgen vervolgens:

```
: cirkelopp DUP * *pi ; ok     ( π maal kwadraat van de straal )
45 cirkelopp . 6361 ok
```

De twee laatste bewerkingen voor 'gemengde nauwkeurigheid' zijn in feite basisbewerkingen, waarmee alle andere vermenigvuldigings- en delings-operaties (inclusief de vorige twee) gedefinieerd zijn. Desgewenst kunnen we er nog meer operaties mee definiëren, die niet in het standaard systeem zitten.

Laten we als voorbeeld aannemen dat we het dubbele nauwkeurigheid product van twee dubbele nauwkeurigheid getallen vaak nodig hebben. Daartoe vermenigvuldigen we met behulp van {U*} de twee helften van beide dubbele nauwkeurigheid getallen met elkaar en combineren dan de deelresultaten zoals in figuur 8.1 aangegeven:

```
                    a     b
                    c     d
                  ------------ *
                  d*be  d*bt
            d*ae  d*at    0
            c*be  c*bt    0
      c*ae  c*at    0     0
  ------------------------------ +
    p     q     r     s
  ------------------------------
```

Figuur 8.1 Dubbele nauwkeurigheid vermenigvuldiging

Hierin is het 32 bits getal ab (a voor de eerste 16 bits, b voor de tweede 16 bits) vermenigvuldigd met het 32 bits getal cd (c voor de eerste 16 bits, d voor de tweede 16 bits) met het 64 bits resultaat pqrs. Elk van de vier deelvermenigvuldigingen produceert een 32 bits deelresultaat, waarvan de eerste helft is aangegeven met de letter e, en de tweede helft met de letter t. Als we vooraf weten dat het resultaat slechts 32 bits lang is, zijn alleen de eerste drie vermenigvuldigingen nodig en hoeven p en q niet bepaald te worden.

Het eenvoudigste programma voor het laatste geval krijgen we door het invoeren van 4 variabelen voor de 16 bits helften a, b, c en d:

```
VARIABLE a ok                ( eerste getal, eerste helft )
VARIABLE b ok                ( eerste getal, tweede helft )
VARIABLE c ok                ( tweede getal, eerste helft )
VARIABLE d ok                ( tweede getal, tweede helft )
: D* a ! b ! c ! d !
     d @ b @ U*
     d @ a @ U* DROP +
     c @ b @ U* DROP + ; ok

6000. 12000. D* D. 72000000 ok
-5000004. 2. D* D. -10000008 ok
```

Hoewel deze oplossing niet zo snel is als een zonder variabelen (die echter veel stapel manipulaties zou vergen) heeft die het voordeel dat het schrijven en begrijpen ervan eenvoudig is! Merk ook op dat het teken van het resultaat automatisch correct is omdat we in feite de eerste 32 bits 'weghakken'.

Alvorens dit onderwerp te verlaten, kan worden opgemerkt dat op deze manier zonder veel extra moeite een 64 bits product van twee 32 bits getallen is te programmeren en dat zo'n programma weer te gebruiken is voor nog nauwkeuriger vermenigvuldigingen.

8.3 De terugkeer-stapel voor zeer snelle programma's

In paragraaf 5.2 is al opgemerkt dat de terugkeer-stapel binnen een colon-definitie als 'extra-hulp' gebruikt kan worden. In FORTH zijn drie bewerkingen voor het bewerken van de terugkeer-stapel:

>R	(n →)	Breng n over van de gewone stapel naar de terugkeer-stapel.
R>	(→ n)	Breng n van de terugkeer-stapel over naar de gewone stapel.
R@	(→ n)	Kopieer n van de terugkeer-stapel naar de gewone stapel. (Hetzelfde als {I} bij de meeste systemen.)

Deze bewerkingen kunnen zeer nuttig zijn, maar moeten zeer voorzichtig worden gebruikt. Let er in het bijzonder op dat:

i) Een colon-definitie uiteindelijk de terugkeer-stapel niet verandert.
ii) De terugkeer-stapel mag in DO-lussen worden gebruikt, mits de index- en limiet-waarden op de terugkeer-stapel niet veranderen (tenzij dat de bedoeling is, zoals in {LEAVE}!).

Laten we als voorbeeld aannemen dat we een opdracht nodig hebben om 1 op te tellen bij het vierde getal op de stapel, zonder de drie getallen daarboven te verknoeien. Met behulp van {>R} kunnen we de bovenste drie getallen tijdelijk overbrengen naar de terugkeer-stapel, dan 1 optellen bij het vierde getal en vervolgens met {R>} de drie getallen weer terugbrengen naar de gewone stapel:

```
: vierde+1 >R >R >R 1+ R> R> R> ; ok
10 20 30 40 vierde+1 . . . . 40 30 20 11 ok
```

De stapelbeschrijving voor deze bewerking is dus:

```
vierde+1        (n1 n2 n3 n4 → n1+1 n2 n3 n4)
```

waarbij de terugkeer-stapel uiteindelijk niet is veranderd. We hadden in dit voorbeeld het gebruik van de terugkeer-stapel kunnen voorkomen:

```
: vierde+1 4 ROLL 1+ 4 ROLL 4 ROLL 4 ROLL ;
```

maar dit is niet alleen veel langer, maar ook veel langzamer omdat {ROLL} een ingewikkelde bewerking is. Als {vierde+1} vaak moet worden uitgevoerd, is de snellere oplossing met de terugkeer-stapel te verkiezen.

Nu volgen nog twee handige opdrachten voor het werken met dubbele-nauwkeurigheid-getallen. Deze opdrachten gebruiken eveneens de terugkeer-stapel voor tijdelijke opslag:

```
: 2SWAP >R ROT ROT R> ROT ROT ;     ( verwissel twee getallen )
: 2OVER 2SWAP 2DUP >R >R 2SWAP R> R> ; ( dupliceer tweede getal
                                        boven op de stapel )
```

(de definitie van {2DUP} staat in paragraaf 8.1.)

Ook handig bij bepaalde toepassingen is de 'gemengde-nauwkeurigheid-deling':

```
: M/MOD >R 0 R@ U/MOD R> SWAP >R  U/MOD R> ;
```

De stapelbeschrijving hiervan is: (ud un → unrest udquotient)

8.4 Afdrukken met indeling

Voor veel echte toepassingen is de met {.} geproduceerde uitvoer niet acceptabel omdat getallen niet met een zekere indeling ('opmaak') zijn afgedrukt. Bijvoorbeeld een datum als 20/01/83 of een prijs als f 2.99. FORTH kent enkele bewerkingen om een dergelijke opmaak op een handige en goed leesbare manier te specificeren. Hier volgt een kort overzicht van de mogelijkheden:

<#	Begin een nieuwe 'opgemaakte' getallenstring.
#	Zet het volgende cijfer van het getal in die string.
#S	Zet verdere significante cijfers in die string.
HOLD	Zet het karakter dat bovenop de stapel staat in die string.
SIGN	Zet zo nodig een min-teken in die string.
#>	Beëindig die string, die nu afgedrukt kan worden.

(Omdat het effect op de stapel van deze woorden niet bekend hoeft te zijn voor hun gebruik, volgt de beschrijving daarvan pas later aan het einde van dit hoofdstuk.)

Met deze FORTH-woorden bouwen we een getal (als string), teken voor teken, op; precies zoals we het op papier willen laten afdrukken. Deze woorden (<#, #S, enz.) drukken zelf niets af; zij geven alleen aan hoe een getal moet worden afgedrukt. De zo opgebouwde string wordt met de uitvoerbewerking {TYPE} (uit het vorige hoofdstuk) afgedrukt.

Hoe dit werkt kunnen we het best met een voorbeeld laten zien. In het volgende demonstreren we het afdrukken van een prijs:

```
DECIMAL ok
: .f  <# # # 46 HOLD #S 70 HOLD #> TYPE SPACE ; ok
1234. .f  f12.34 ok
```

De reeks van bewerkingen voor het afdrukken van het dubbele-nauwkeurigheid-getal 1234 houdt het volgende in:

i) Door {<#} wordt een speciale buffer (in feite PAD achterstevoren) klaargezet voor het opnemen van karakters.

ii) De eerste {#} zet het *laatste cijfer* van het getal (dus 4) om in de corresponderende ASCII-code (met het huidige - decimale - grondtal) en plaatst dit in de buffer.

iii) De tweede {#} zet het volgende cijfer (3) om en plaatst het in de buffer.

iv) Dan zet {46 HOLD} de waarde 46 (de ASCII-code voor een decimaal punt) in de buffer.

v) Door {#S} worden de overige significante cijfers van het getal (2 en dan 1) omgezet en in de buffer geplaatst.

vi) Door {70 HOLD} wordt de ASCII-waarde voor *f* in de buffer gezet.

vii) Door {#>} wordt de nu complete string beëindigd en het adres en aantal karakters daarvan op de stapel gezet voor {TYPE}.

viii) Door {TYPE SPACE} wordt de string in de buffer, beginnend met het laatst toegevoegde karakter (*f*), afgedrukt en daarna volgt een spatie.

Belangrijk is te onthouden dat een opgemaakt getal van achter naar voor in de buffer wordt opgebouwd en dat de opmaak-bewerkingen in samenhang met dubbele-nauwkeurigheid-getallen zijn ontworpen. Het laatste is prettig omdat we dan tien decimale cijfers kunnen gebruiken, terwijl de vijf cijfers voor enkele-nauwkeurigheid-getallen vaak niet voldoende zijn.

Voor de omzetting mag het dubbele-nauwkeurigheid-getal geen teken hebben. Willen we ook negatieve getallen opgemaakt afdrukken, dan moeten die vóór {<#} positief worden gemaakt (met {DABS}), nadat het teken is 'onthouden' door {SIGN}. Het eenvoudigst gaat dat met de opdrachten {SWAP OVER DABS} voorafgaand aan {<#}. Met een nieuwe definitie van {.*f*} kunnen we dan zowel debet als credit bedragen afdrukken:

```
: .f  SWAP OVER DABS
      <#  # # 46 HOLD  #S   70 HOLD SIGN  #>
      TYPE SPACE ; ok

-12345.  .f  -f123.45 ok
```

Het effect van {SWAP OVER DABS} op het dubbele nauwkeurigheid getal (met teken) op de stapel is als volgt:

```
SWAP      (dt de → de dt)      Verwissel de twee helften van d.
OVER      (de dt → de dt de)   Dupliceer eerste helft van d.
DABS      (de d  → de |d|)     Maak d positief.
```

Verderop in de colon-definitie wordt door {SIGN} het teken van 'de' (dat gelijk is aan dat van d) gebruikt om hetzij een min-teken in de buffer te zetten, hetzij niets te doen.

Een laatste verfijning van {.f} moet tenslotte bewerkstelligen dat de decimaal punten van getallen in een kolom recht onder elkaar komen en dat de breedte van de kolom bepaald wordt door het getal op de top van de stapel:

```
: .f  >R                                  ( kolombreedte redden )
      SWAP OVER DABS                      ( voor negatieve getallen )
      <#  # #  46 HOLD  #S  70 HOLD #>    ( vul de buffer )
      R  OVER - SPACES+                   ( druk beginspaties af )
      TYPE SPACE ; ok                     ( en het getal )

-0.01 CR 10 .f                            ( kolombreedte van 10 )
    -f0.01 ok

123.45 CR 10 .f
   f123.45 ok
```

Hierin gebruikten we het aantal karakters, verkregen met {#>}, om de vereiste beginspaties af te drukken voordat we met {TYPE} het getal zelf op papier zetten.

Tot slot volgen nu nog enkele goed bruikbare definities, o.a. voor een datum, zoals vermeld in het begin van deze paragraaf:

```
: .datum <# # # 47 HOLD # # 47 HOLD # #  47 HOLD #>   TYPE ;

( druk een dubbele nauwkeurigheid getal af en dan een spatie )
: D.      SWAP OVER DABS          ( d → )
          <#  #S SIGN #>
          TYPE SPACE ;

( druk een dubbele nauwkeurigheid getal af in een kolom met
  breedte n)
: D.R     >R                      ( d n → )
          SWAP OVER DABS <#  #S SIGN  #>
          R>  OVER - SPACES TYPE ;
```

```
( druk een getal n1 af in een kolom met breedte n2 )
: .R      >R S->D R> D.R ;          ( n1 n2 → )
```

Het woord {S->D} in de laatste definitie zorgt voor de omzetting van een enkele-nauwkeurigheid-getal in een dubbele-nauwkeurigheid-getal. Als je systeem dit woord niet kent, kun je het zelf definiëren met:

```
: S->D    DUP 0<  IF  -1 ELSE 0 THEN ;
```

8.5 Rekenen met vaste komma

We hebben nu alles voorhanden om een 'programma-pakket' te maken voor het rekenen met een vaste komma. Onder getallen met vaste komma verstaan we getallen waarin de decimaal punt (in computertalen wordt de punt gebruikt; wij zelf gebruiken de komma en spreken over getallen met een 'vaste' of 'zwevende' komma) een vaste positie heeft (wij zeggen: getallen met evenveel cijfers achter de komma!). We zullen zien dat we met dergelijke 'vaste-komma' getallen net zo kunnen rekenen als met dubbele-nauwkeurigheid-getallen. De term 'vaste-komma' slaat eigenlijk alleen op de manier waarop we deze getallen invoeren en laten afdrukken; niet op de manier waarop de computer er mee rekent.

Neem eens aan dat we besluiten te gaan werken met getallen met vier cijfers achter de komma (in FORTH: achter de punt). Het bereik van zulke getallen is dan:

```
(+ of -) 0.0001  t/m  (+ of -) 99999.9999
```

Zulke getallen kunnen zonder enig bezwaar met FORTH ingevoerd worden. Laten we dit eens doen:

```
45.8045   19.0030  D+ D.   648075 ok
```

FORTH drukt als antwoord 648075 af. De decimaal punt wordt door FORTH niet gezien als een 'echte' decimaal punt maar als een teken (zie p. 78) dat de getallen dubbele-nauwkeurigheid-getallen zijn.

Voordat FORTH het 'juiste' antwoord kan afdrukken moet de uitkomst 'opgemaakt' worden. Met behulp van de hierboven besproken methoden zullen we een afdrukprogramma schrijven voor het afdrukken van getallen met vier cijfers achter de komma. Dit programma {V.} gebruiken we dan in plaats van {D.}. Het programma ziet er zo uit:

```
: V.      SWAP OVER DABS
          <#  # # # #  46 HOLD  #S SIGN  #>
          TYPE SPACE ; ok

0.0005 0.0100 D+ V. 0.0105 ok
100.0000 0.0001 D- V. 99.9999 ok
```

Het lijkt er op dat we nu onze vaste komma optelling en aftrekking hebben, maar met een groot gebrek; we moeten altijd vier cijfers na de decimaal punt intikken! Doen we dit niet, dan kunnen we gekke fouten krijgen, zoals bijvoorbeeld:

```
100.1 0.25 D+ V. 0.1026 ok
```

Om dit weer in orde te brengen, hebben we blijkbaar een bewerking nodig die getallen, hoe ze ook ingetikt mogen zijn, in de juiste vorm (d.w.z. met vier cijfers achter de komma) op de stapel zet. Het getal 100.8 moet bijvoorbeeld opgevat worden als 1008000 om overeenstemming te krijgen met de gekozen notatie voor vaste komma getallen. De te gebruiken vermenigvuldigingsfactor hangt dan af van het aantal na de decimaal punt ingetikte cijfers. Dit aantal is te halen uit de systeem-variabele {DPL} (zie paragraaf 8.1). In de volgende definitie wordt {DPL} gebruikt om de macht van tien vast te stellen waarmee een getal uit de invoer vermenigvuldigd moet worden om het goede resultaat te krijgen:

```
: SF  DPL @ 0< IF S->D 0 DPL !  ( zet zo nodig om in 32 bits )
                THEN
      DPL @ DUP 4 <
          IF 4 SWAP DO 10. D*   ( schaling van getal )
                    LOOP
          ELSE 4 >
            IF ." Ontoelaatbaar"
               2DROP
            THEN
          THEN ;
```

In deze definitie zijn nog enkele verfijningen aangebracht. Een daarvan is dat ook enkele-nauwkeurigheid-getallen worden geschaald (die {DPL} de waarde -1 geven). De eerste 'IF' verandert deze namelijk in dubbele nauwkeurigheid getallen zonder cijfers achter de decimaal punt, zodat ze correct geschaald worden door het daarop volgende programma.Een andere verfijning is dat getallen met meer dan vier cijfers achter de decimaal punt de foutmelding "Ontoelaatbaar" zullen veroorzaken, maar geen andere getallen op de stapel zullen verknoeien.

We kunnen nu bruikbare berekeningen uitvoeren, zoals bijvoorbeeld:

```
0.04 SF ok
0.1 SF D+ ok
0.567 SF D+ ok
0.0001 SF D+  ok
10 SF D+ ok
V. 10.7071 ok
```

Dit pakket zouden we nu zo nodig kunnen uitbreiden met andere bewerkingen. Een vermenigvuldiging wordt helemaal correct uitgevoerd met de {D*}-operatie van paragraaf 8.2 behalve dat het resultaat 10000 maal te groot is (door de plaats van de decimaal punt). Dit probleem is eenvoudig op te lossen met een opdracht die een getal door 10000 deelt:

```
: /10000 DUP >R           ( teken op de terugkeer-stapel )
         DABS             ( absolute waarde van getal )
         10000 M/MOD      ( gemengde nauwkeurigheid deling )
         R> 0< IF DNEGATE THEN  ( quotient teken aanpassen )
         ROT DROP ;       ( en rest verwijderen )
```

Samen met {D*} krijgen we voor een vaste komma vermenigvuldiging:

```
: V*  D* /10000 ; ok
0.456 SF 20 SF V* V. 9.1200 ok
-0.05 SF 0.6 SF V* V. -0.0300 ok
```

Natuurlijk zal {V*} niet goed werken voor heel grote getallen, waarbij het product buiten het bereik van dubbele nauwkeurigheid getallen kan komen. Om deze onvolkomenheid op te vangen moet een dubbele-nauwkeurigheid 'vermenigvuldig en deel'-opdracht (zoiets als {D*/}) geschreven worden waarin een 64-bits tussenresultaat ontstaat.

Tenslotte kan nog opgemerkt worden dat, aangezien het rekenen met vaste komma in feite gebeurt met gehele getallen, dit rekenen heel snel gaat en zeker veel sneller dan het rekenen met getallen met een zwevende komma.

8.6 Samenvatting

De volgende standaard FORTH-79 opdrachten zijn in dit hoofdstuk besproken:

Stapel manipulatie

`>R` (n →)
Verplaats `n` naar de terugkeer-stapel voor tijdelijke opslag. Bij iedere `>R` behoort binnen elke structuur van een colon-definitie een corresponderende `R>`.

`R>` (→ n)
Verplaats `n` van de terugkeer-stapel naar de gewone stapel.

`R@` (→ n)
Kopieer het getal op de terugkeer-stapel op de gewone stapel.

Vergelijkingsbewerking

`D<` (d1 d2 → b)
`b` is alleen true als `d1` kleiner is dan `d2`.

Rekenbewerkingen

`D+` (d1 d2 → dsom)
Optelling van twee dubbele nauwkeurigheid getallen.

`DNEGATE` (d → -d)
Tekenomkeer van een dubbele nauwkeurigheid getal.

`*/` (n1 n2 n3 → nquotient)
Berekening van `n1 * n2 / n3` met een 32 bits tussenresultaat.

`*/MOD` (n1 n2 n3 → nrest nquotient)
Als het voorgaande, maar ook de rest met het teken van `n1`.

U* (un1 un2 → udproduct)
Dubbele nauwkeurigheid product van twee enkele nauwkeurigheid getallen zonder teken.

U/MOD (ud un → urest uquotient)
Deel dubbele nauwkeurigheid getal (zonder teken) door enkele nauwkeurigheid getal en zet rest en quotient (alles zonder teken) op de stapel.

Afdrukbewerkingen en diversen

<# (→)
Begin van een getalomzetting met indeling.

(ud1 → ud2)
Maak voor het dubbele nauwkeurigheid getal zonder teken ud1 het volgende ASCII karakter en voeg het aan de cijferbuffer toe. Het getal ud2 is het quotient ud1/ BASE voor de bepaling van het volgende cijfer. Gebruik dit tussen <# en #>.

#S (ud → 0 0)
Zet de resterende significante cijfers van ud om en voeg ze aan de cijferbuffer toe. Op de stapel blijven twee nullen achter. Was ud aanvankelijk nul, voeg dan één nul toe aan de cijferbuffer. Gebruik dit tussen <# en #>.

HOLD (kar →)
Zet het karakter op de stapel in de cijferbuffer. Gebruik dit tussen <# en #>.

SIGN (n ud → ud)
Zet het ASCII min-teken in de cijferbuffer als n negatief is. Gebruik dit tussen <# en #>.

#> (ud → adres n)
Einde van getalomzetting. Laat het adres van de cijferbuffer en het aantal karakters daarin achter op de stapel voor {TYPE}.

79-STANDARD (→)
Verifieer dat het systeem voldoet aan de FORTH-79 norm.

9 UITBREIDEN VAN FORTH

Naast woorden als {*}, {LIST} en {kwadraten} (zie p. 58) die alle een bepaalde reeks acties tot gevolg hebben zijn er ook FORTH-woorden, zoals {VARIABLE} en {CREATE}, die naast acties ook zelf nieuwe woorden definiëren. We kunnen deze woorden dan ook definiërende-woorden noemen. Een van de opmerkelijkste eigenschappen van FORTH is dat wij zelf zulke definiërende woorden kunnen maken. We kunnen hiermee een nieuw FORTH-woord aan het Woordenboek toevoegen waarbij tevens wordt vastgelegd welke acties ondernomen moeten worden bij het uitvoeren van dit woord. Dit geeft ons de mogelijkheid nieuwe gegevensstructuren als 'stringvariabelen', 'meerdimensionale arrays' of zelfs een mengeling van gegevensstructuren te definiëren. Ook kunnen we nieuwe vertaalwoorden (die tijdens de vertaalfase worden uitgevoerd, zoals IF en DO) maken om zo nieuwe programmastructuren te kunnen gebruiken.

9.1 Het definiëren van nieuwe definiërende woorden

We hebben al gezien hoe we arrays kunnen declareren met behulp van de woorden {CREATE} en {ALLOT}, gevolgd door een speciale colon-definitie om het adres van een benodigd array-element te berekenen. Om het geheugen wat op te frissen geven we eerst de declaratie van een array met tien elementen:

```
CREATE x 20 ALLOT          ( definitie van het array )
: X x SWAP 2 * + ;         ( adresberekening )
```

Hoewel deze methode volkomen bevredigend is, heeft hij het nadeel dat voor ieder nieuw array deze twee FORTH regels (of iets soortgelijks) herhaald moeten worden.

We zouden baat hebben bij een totaal nieuw definiërend woord {ARRAY} met dezelfde uitwerking als de voorgaande twee FORTH regels, maar toepasbaar voor verschillende array namen. Zo'n definiërend woord kunnen we inderdaad maken met een speciale colon-definitie en het woord {DOES>}:

```
: ARRAY
        CREATE 2 * ALLOT      ( maak nieuwe toevoeging )
        DOES> SWAP 2 * + ;    ( effect bij uitvoering )
```

Het declareren van een array X met tien elementen gaat dan met:

```
10 ARRAY X
```

en van een array Y met 20 elementen met:

```
20 ARRAY Y
```

Voor het verwijzen naar een element van een array, moeten we voor de array naam en volgordenummer (tellend vanaf 0) van het element zetten. Bijvoorbeeld:

```
4 X ?          ( druk het vijfde element van X af )
15 Y ?         ( druk het zestiende element van Y af )
```

De werking van {ARRAY} is niet zo vanzelfsprekend. We zullen daarom stap voor stap aangeven wat er gebeurt als {ARRAY} wordt uitgevoerd. Allereerst zal het woord volgend op ARRAY (X en Y in bovenstaande voorbeelden) als nieuw woord in het Woordenboek worden genoteerd. Het getal bovenop de stapel, voorafgaande aan ARRAY (10 en 20) in bovenstaande voorbeelden) zal vervolgens door {ALLOT} met 2 worden vermenigvuldigd; dit is nodig voor het reserveren van de vereiste geheugenruimte. Het woord {DOES} (na ALLOT) in de colon-definitie wordt alleen gebruikt in nieuwe 'definiërende' woorden. Achter DOES> wordt namelijk aangegeven wat er gebeurt als de woorden {X} en {Y}, die met het definiërende woord {ARRAY} gedeclareerd zijn, als opdrachten in een stuk programma worden uitgevoerd.

Zonder het gedeelte {DOES> SWAP 2 * +} in de definitie van {ARRAY} zouden nieuwe, met {ARRAR} gedeclareerde, woorden hetzelfde effect hebben als met {CREATE} gedefinieerde woorden; namelijk het op de stapel zetten van het (eerste) adres van de gereserveerde geheugenruimte. Doordat {DOES>} in de definitie staat zullen de opdrachten achter DOES> worden uitgevoerd met dit adres bovenop de stapel. Daarom zal:

```
4 X
```

tot gevolg hebben dat {DOES> SWAP 2 * +} wordt verwerkt met bovenop de stapel het adres van het eerste element van X en daaronder het getal 4. Het resultaat daarvan is dat op de stapel het adres van het vijfde element (het tellen begint bij 0!) terecht komt, zoals hierboven al gebruikt.

Belangrijk is te onthouden dat de opdrachten achter {DOES>} niet uitgevoerd worden wanneer {ARRAY} wordt uitgevoerd, maar pas wanneer de met {ARRAY} gedefinieerde woorden uitgevoerd worden.

De algemene structuur van een colon-definitie voor het definiëren van een nieuw definiëren woord is:

```
: nieuw_definiërend_woord
          CREATE          ( tijdens vertaling uit te voeren )
          DOES>           ( tijdens verwerking uit te voeren ) ;
```

Tot slot van deze paragraaf volgt nu nog een greep uit de definities van 'standaard' definiërende woorden:

```
: VARIABLE CREATE 2 ALLOT ;                ( 16 bits variabele )
: CONSTANT CREATE , DOES> @ ;              ( 16 bits constante )
```

Bij de meeste systemen zijn {VARIABLE} en {CONSTANT} wel als basis machine-opdrachten gedefinieerd, maar we zouden bijvoorbeeld {VARIABLE} zodanig kunnen herdefiniëren dat die door definitie tevens een beginwaarde krijgt (zoals in vroegere FORTH normen):

```
: VARIABLE CREATE , ;
```

Met {C,} in plaats van {,} worden alleen de laatste 8 bits (1 byte) van het getal op de stapel in het Woordenboek gezet. Kent je systeem dit woord {C,}, dan kun je daarmee variabelen en constanten van 1 byte definiëren:

```
: CVARIABLE CREATE 1 ALLOT ;               ( 8 bits variabele )
: CCONSTANT CREATE C, DOES> C@ ;           ( 8 bits constante )
```

Tenslotte voor dubbel-nauwkeurigheid-variabelen en constanten:

```
: 2VARIABLE CREATE 4 ALLOT ;               ( 32 bits variabele )
: 2CONSTANT CREATE , ,                     ( 32 bits constante )
            DOES> DUP 2+ @ SWAP @ ;
```

Hierbij passende opdrachten om dubbele nauwkeurigheid getallen op de stapel te zetten of ze er af te halen, zijn eenvoudig te definiëren:

```
: D! DUP >R ! R> 2+ ! ;                    ( d adres → )
: D@ DUP @ >R 2+ @ R> ;                    ( adres → d)
```

9.2 Het laatste woord over ARRAYs

Heel eenvoudig kunnen nog enkele nuttige verfraaiingen opgenomen worden in het nieuwe definiërende woord {ARRAY} van de vorige paragraaf. Een daarvan is een controle op een illegale indexwaarde, een andere is de array-elementen van 1 af te tellen en niet van 0 af. Dit is in de volgende definitie verwerkt (de regelnummers zijn alleen toegevoegd voor latere verwijzingen):

```
0  : ARRAY
1        CREATE DUP ,                      ( berg array-grootte op )
2        2 * ALLOT                         ( en reserveer ruimte )
3        DOES>
4        SWAP 1- SWAP                      ( index vanaf 1 )
5        OVER OVER                         ( dupliceer index en adres )
6        @ U< NOT IF                       ( test indexbereik )
7                  ." Illegale
                        index"
8                  QUIT
9                THEN
10       2 +
11       SWAP 2 * + ;                      ( bereken adres )
```

Deze nieuwe versie van {ARRAY} kan net als de oude worden gebruikt:

```
20 ARRAY tabel ok          ( definieer een array met 20 elementen )
0 tabel Illegale index     ( index buiten de grenzen )
21 tabel Illegale index
-1 5 tabel ! ok            ( vijfde element wordt -1 )
```

Tijdens de vertaling wordt de afmeting van het array nu door {DUP} bewaard, zodat deze tijdens de verwerking kan worden gebruikt voor het controleren van de indexwaarde.

In het verwerkingsdeel zijn drie stukken te onderscheiden. De eerste regel na {DOES>} verlaagt de index (onder de top van de stapel) met 1, hetgeen betekent dat als we element 1 van het array noemen, we in feite het nulde element krijgen.

Regels 5 t/m 9 controleren de indexwaarde. Interessant is hier het gebruik van {U<}, dat er voor zorgt dat ook negatieve indexwaarden aanleiding geven tot een foutmelding omdat negatieve getallen, opgevat zonder teken, er uit zien als grote positieve getallen. Daardoor voorkomen we twee aparte vergelijkingsbewerkingen voor getallen met teken.

Regel 10 past het adres op de top van de stapel aan om de opgeslagen afmeting van het array over te slaan. Regel 11 berekent het adres van het gewenste array-element zoals in de eerdere definitie van {ARRAY}.

Regels 4 t/m 11 worden natuurlijk iedere keer uitgevoerd wanneer een array-element, gedefinieerd met {ARRAY}, wordt genoemd. Dit kost tijd voor het controleren van de indexwaarde. Omdat dit alleen nodig is tijdens het ontwikkelen van een programma, is het gebruikelijk om deze controle weer te verwijderen wanneer het programma geheel getest is; de verwerking gaat dan sneller. Van bovenstaande definitie blijft dan over:

```
: ARRAY CREATE 2 * ALLOT
        DOES> SWAP 1- 2 * + ;
```

Tot slot van deze paragraaf volgt een definitie van {2DARRAY}, een definiërend woord voor het declareren van twee-dimensionale arrays:

```
: 2DARRAY CREATE DUP ,          ( bewaar tweede index )
          * 2 * ALLOT           ( reserveer array-ruimte )
          DOES> ROT             ( i1 naar stapeltop )
          OVER @ *              ( maal bewaarde index )
          ROT +                 ( tel i2 er bij op )
          2 * + 2 + ;           ( bereken adres van i1,i2 )
```

Hierin hebben i1 en i2 betrekking op de indices van een array-element:

```
4 4 2DARRAY blok                ( definieer een 4 × 4 array 'blok' )
-1 0 0 blok !                   ( element 0,0 wordt -1 )
```

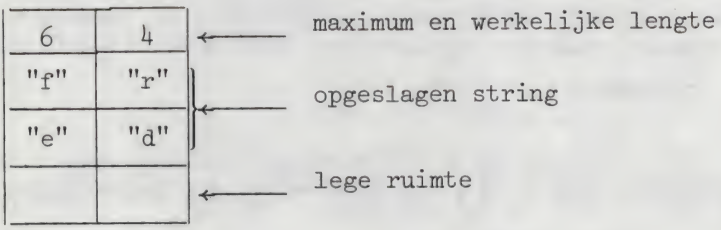
Dit array bevat 16 elementen met indices van 0,0 tot 3,3. Een goede manier om de elementen achtereenvolgens op te halen, maakt gebruik van geneste DO-lussen:

```
: blokafdruk                    ( druk het hele blok af )
    4 0 DO                      ( ga van 0 naar 3 )
      4 0 DO                    ( idem )
        J I blok .              ( druk element I,J af )
          LOOP CR
        LOOP ;
```

9.3 Een STRING variabele

Een andere kandidaat voor een nieuw definiërend woord is de STRING variabele. Hiermee kunnen we de in hoofdstuk 7 genoemde beperkingen ten aanzien van het werken met strings wegnemen. Deze beperking geldt vooral het gebruik van een vast buffergebied voor alle string bewerkingen. Ook kunnen we hiermee bewerkingen maken waarmee op een fraaie manier strings kunnen worden samengevoegd of met elkaar kunnen worden vergeleken (zoals dat ook in BASIC mogelijk is).

Voor het ontwerpen van een definiërend woord voor strings moeten we eerst nagaan welke grootheden we allemaal door een string variabele willen laten beschrijven. In de eerste plaats natuurlijk een string van karakters. Maar verder zou het gemakkelijk zijn wanneer een string variabele ook de maximum toegelaten lengte en de werkelijke lengte van de karakterstring zou bevatten. Figuur 6.1 laat zien hoe een string variabele, gedefinieerd voor maximaal 6 karakters, de karakterstring "fred" van 4 karakters opslaat.



Figuur 9.1 Een structuur voor een string variabele

De maximum lengte wordt gegeven in de definitie van de string variabele en gebruikt om 'string overloop' te controleren. De werkelijke lengte zal string manipulatie vereenvoudigen, bijvoorbeeld voor de opdracht {TYPE} voor het afdrukken van een string.

In overeenstemming met het voorgaande is een definitie voor {STRING} (zie pagina 92 voor {C,}):

```
: STRING
     CREATE DUP C,          ( maximum lengte )
            0 C,            ( werkelijke lengte eerst 0 )
            ALLOT           ( reserveer geheugenruimte )
     DOES>
            2+              ( beginadres van string )
            DUP 1- C@ ;     ( werkelijke lengte )
```

Voor een string variabele A$ met ruimte voor een string van 20 karakters, tikken we in:

```
20 STRING A$
```

Door het declareren van A$ blijven twee waarden op de stapel achter, het maximum aantal karakters op de top en het beginadres van de string daaronder. Dit is precies wat de standaard afdrukbewerking {TYPE} voor het afdrukken van een string nodig heeft, zodat we A$ kunnen afdrukken met:

```
A$ TYPE
```

We hebben hier eigenlijk pas wat aan als we invoerslierten (karakterstrings) aan stringvariabelen kunnen toekennen. Hiervoor hebben we twee bewerkingen nodig, {INPUT$} voor invoer via het toetsenbord bij 'vraag en antwoord' programma's, en {PUT$} voor het maken van strings in een programma. Aangezien deze twee definities bijna identiek blijken te zijn, wordt bij de tweede het meeste commentaar weggelaten.

```
: INPUT$ DROP 1-                        ( adres van karakteraantal )
         DUP 1- C@                      ( maximum lengte )
         CR ." ? " QUERY                ( string van toetsenbord )
         1 WORD                         ( naar HERE )
         HERE C@ <                      ( werkelijke stringlengte )
         IF ." Te lange string"         ( foutmelding )
            DROP QUIT                   ( stapel schoon en ophouden )
         THEN
         HERE DUP C@ 1+                 ( adres en bytes voor )
         ROT SWAP CMOVE ;               ( plaatsing in string variabele )

: PUT$   DROP 1-
         DUP 1- C@                      ( haal string volgend )
         36 WORD                        ( op PUT$ )
         HERE C@ <
         IF ." Te lange string"
            DROP QUIT
         THEN
         HERE DUP C@ 1+
         ROT SWAP CMOVE ;
```

In {PUT$} is als slotkarakter voor {WORD} de ASCII-waarde voor het karakter "$" genomen. De invoer van strings met {PUT$} moet dus eindigen met "$" en mag spaties bevatten. Hier volgen enkele voorbeelden van het gebruik van {INPUT$} en {PUT$}:

```
20 STRING A$ ok                    ( string A$ met 20 karakters )
A$ INPUT$                          ( invoer van toetsenbord )
? test-string ok                   ( na indrukken van 'return' )
A$ TYPE test-stringok              ( afdrukken van string )
A$ INPUT$
? deze string is te lang Te lange string
A$ PUT$ nog een test$ ok           ( invoer van invoersliert )
A$ TYPE nog een testok             ( afdrukken van string )
```

Een nuttige aanvulling van onze woordenlijst voor het behandelen van strings is een bewerking voor het vergelijken van twee strings:

```
: Ongelijk OVER OVER               ( dupliceer lengte en adres2 )
           + SWAP
           DO DROP 1+ DUP 1- C@    ( karakters van string1 )
              I C@ - DUP           ( ongelijk? )
                IF
                  DUP ABS / LEAVE
                THEN
           LOOP SWAP DROP ;
```

De bewerking {Ongelijk} zal twee strings van gelijke lengte met elkaar vergelijken en heeft als stapel-beschrijving:

```
Ongelijk          (adres1 n adres 2 → vlag)
```

De twee strings, beginnend op adres1 en adres2 worden met elkaar vergeleken en zetten een vlag op de stapel die de waarde 'false' heeft als de strings gelijk zijn, de waarde 'positieve true' als string1 groter is dan string2 en de waarde 'negatieve true' als string1 kleiner is dan string2. Dit kan weer worden gebruikt in de volgende definitie:

```
: $= ROT OVER =                    ( strings met dezelfde lengte? )
     IF
       SWAP Ongelijk NOT           ( onderzoek ze verder )
     ELSE
       DROP DROP DROP 0            ( anders false )
     THEN ;
```

Een interessante toepassing van {$=}, die weer eens laat zien dat je in FORTH geen 'GOTO' nodig hebt, is een colon-definitie om de gebruiker te vragen of hij verder wil gaan of niet. In BASIC zien we deze constructie vaak in spelprogramma's, zoals in het nu volgende voorbeeld:

```
10 PRINT "Wil je verder gaan (ja/nee)";
20 INPUT A$
30 IF A$="nee" THEN END
40 IF A$="ja" THEN 60
50 GOTO 10
60 ....
```

In FORTH doen we dit zo:

```
10 STRING antwoord$
2 STRING Ja$     Ja$ PUT$ ja$
3 STRING Nee$    Nee$ PUT$ nee$
: Doorgaan?
     BEGIN ." Wil je doorgaan (ja/nee) "
           antwoord$ INPUT$              ( wacht op antwoord )
           antwoord$ Nee$ $= IF QUIT THEN
           antwoord$ Ja$ $=
     UNTIL ;                             ( herhaling van lus )
```

Door het woord {Doorgaan?} op te nemen kun je een programma tijdens de verwerking laten stoppen na de boodschap: "Wil je doorgaan (ja/nee)". Is het dan ingetikte antwoord "Nee", dan houdt de verwerking definitief op. Is het antwoord "Ja", dan gaat het programma verder. Is het antwoord noch "Ja", noch "Nee", dan wordt de vraag herhaald.

9.4 Zichzelf veranderende gegevensstructuren

Een merkwaardig gevolg van de mogelijkheid om in FORTH nieuwe definiërende woorden te declareren, is dat we 'intelligente' gegevensstructuren kunnen maken. Bijvoorbeeld arrays die automatisch gemiddelden van hun elementen bijhouden of lijsten die zichzelf opnieuw sorteren wanneer een element veranderd wordt.

Voor het eerste van deze voorbeelden beschouwen we een array 'data' met 10 elementen, dat gedefinieerd is met het woord {ARRAY} van paragraaf 9.2. Om het gemiddelde van de elementen van dit array te berekenen, moeten we zijn 10 elementen bij elkaar optellen en de som door 10 delen. Een opdracht om dit te doen is snel geschreven:

```
: gem. 0                          ( gemiddelde is 0 )
       11 1 DO                    ( lus voor 10 elementen )
              I data @ +          ( tel element op )
            LOOP
       10 / ;                     ( en deel door 10 )
```

Als onze FORTH toepassing met zich mee zou brengen dat zo'n gemiddelde vaak berekend moet worden en voor verschillende arrays, dan kunnen we ter vereenvoudiging van het programmeren van die toepassing beter een nieuw definiërend woord {*ARRAY} declareren, met de berekening van het gemiddelde in het {DOES>} gedeelte:

```
: *ARRAY
    CREATE
        DUP ,                   ( bewaar array afmeting )
        0 ,                     ( gemiddelde wordt 0 )
        0 DO 0 , LOOP           ( elementen worden 0 )
    DOES>
        DUP DUP @               ( array afmeting )
        SWAP 4 +                ( naar begin van het array )
        OVER 0 SWAP
        0 DO                    ( voor alle elementen )
            OVER @ +            ( tel element op )
            SWAP 2+ SWAP        ( wijs naar volgende element )
            LOOP
        SWAP DROP SWAP /        ( deel door array afmeting )
        OVER 2+ !               ( bewaar gemiddelde )
        2+ SWAP 2 * + ;         ( bereken adres )
```

Met {*ARRAY} gedefinieerde arrays kunnen net als de met {ARRAY} gedefinieerde worden gebruikt, bijvoorbeeld:

```
10 *ARRAY data ok               ( verzameling van 10 elementen )

10 1 data ! ok                  ( data(1) wordt 10 )
20 2 data ! ok                  ( data(2) wordt 20 )
1000 10 data ! ok               ( data(10) wordt 1000 )

2 data ? 20 ok                  ( druk data(2) af )
```

Dit is precies wat we voor een array met tien elementen en indices van 1 t/m 10 zouden verwachten. Maar:

```
0 data ? 103 ok
```

geeft het gemiddelde van de getallen die op dit moment in het array zijn opgeslagen (door {CREATE} waren oorspronkelijk tien nullen opgeslagen!). Dit gemiddelde wordt iedere keer als het array {data} wordt genoemd, opnieuw berekend en zal dus altijd correct zijn hoeveel veranderingen ook in het array zijn aangebracht. Bijvoorbeeld:

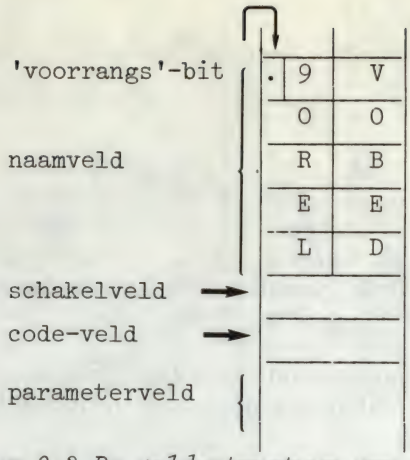
```
870 10 data ! ok                ( verander data(10) in 870 )
50 6 data ! ok                  ( data(6) wordt 50 )

0 data ? 95 ok                  ( nieuwe gemiddelde is 95 )
```

Deze prettige eigenschap hebben alle met {*ARRAY} gedefinieerde arrays!

9.5 Het Woordenboek nader bekeken

Alle toevoegingen aan het Woordenboek hebben dezelfde opslagstructuur. Ieder onderdeel daarvan heet in FORTH een 'veld' en in iedere toevoeging kun je vier velden onderscheiden: het naamveld, het schakelveld, het code-veld en het parameterveld. Figuur 9.2 laat dit zien voor de toevoeging "VOORBEELD" (een variabele, constante of definitie):



Figuur 9.2 De veld-structuur van een toevoeging aan het Woordenboek

Het naamveld bevat eerst het aantal karakters van de opgegeven {naam} en vervolgens de karakters (opgeslagen met ASCII bytes) van die {naam}, die dus de lengte van het naamveld bepaalt. Volgens de FORTH-79 norm is die lengte maximaal 32 bytes, zodat van {naam} tenhoogste de eerste 31 karakters worden vastgelegd. (Raadpleeg echter de documentatie van je eigen systeem, omdat bij veel systemen het naamveld beperkt is tot vier karakters en de naam dus tot drie karakters!)
Van het eerste veld zijn 7 bits gereserveerd voor het aantal karakters, zodat de naam beslist niet meer dan 127 karakters lang mag zijn. De eerste bit van dit veld heet 'voorrangs'-bit, zijn functie wordt later besproken.

Het schakelveld van altijd 16 bits bevat het adres van de vorige toevoeging aan het Woordenboek en wordt gebruikt bij het opsporen van een woord in het Woordenboek. Het code-veld is eveneens 16 bits lang en bevat het adres van de reeks opdrachten die uitgevoerd moet worden voor de desbetreffende {naam}. De aard van deze reeks is afhankelijk van het type definiërend woord, zoals in de volgende tabel is vermeld:

Definiërend woord	*Aard van de reeks opdrachten*
VARIABLE of CREATE	Zet het beginadres van het parameterveld op de stapel.
CONSTANT	Zet de inhoud van het eerste 16 bits parameterveld op de stapel.
:	Voer de opdrachten van de colon-definitie uit met behulp van de in het parameterveld opgeslagen adressen.

De reeks opdrachten is gewoonlijk een reeks instructies in de machinetaal (ter wille van een snelle uitvoering), maar met behulp van {DOES>} kunnen we een eigen code met door {CREATE} ingevoerde (gedefinieerde) woorden verzinnen. Dit zagen we al in het begin van dit hoofdstuk.

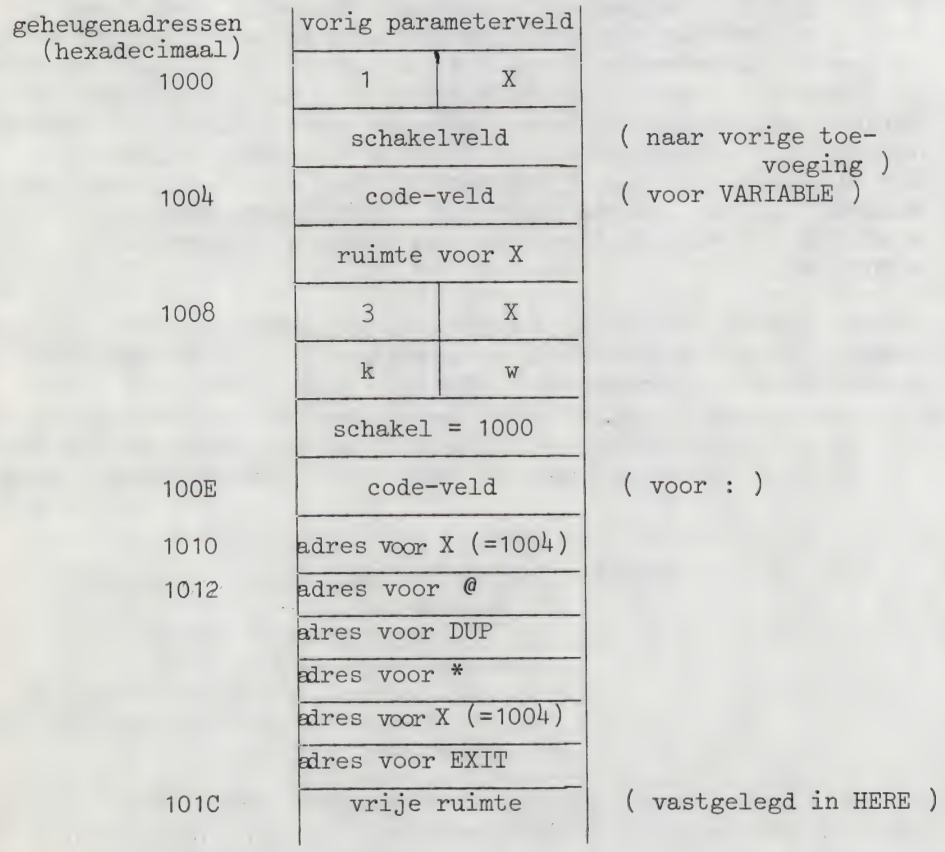
Het laatste veld voor een toevoeging aan het Woordenboek, het parameterveld, kan van alles bevatten. Bij

```
CREATE nul ok
```

is het parameterveld leeg en heeft het ook de lengte 0 (maar evengoed krijgen we met {nul} het adres van het parameterveld). Bij toevoegingen met behulp van {VARIABLE} of {CONSTANT} bestaat het parameterveld uit een 'cel' van 16 bits, waarin de waarde van de variabele of constante wordt opgeslagen. In deze drie gevallen kan het parameterveld langer worden gemaakt met {ALLOT}, {,} of {C,}.

Voor een colon-definitie bevat het parameterveld een lijst van adressen met een adres voor ieder woord in de colon-definitie. In figuur 9.3 is dit weergegeven voor:

```
VARIABLE X
: Xkw X @ DUP * X ! ;                    ( kwadrateer X )
```



Figuur 9.3 Twee toevoegingen aan het Woordenboek

Via het code-veld voor : wordt een 'adres-interpretator' programma aan het werk gezet, dat met behulp van de adressen in het parameterveld de benodigde bewerkingen uitvoert. Wanneer FORTH bijvoorbeeld {Xkw} moet uitvoeren, wordt via het code-veld (op adres 100E) de adres-interpretator geactiveerd. De adres-interpretator zal dan:

i) het eerste adres in het parameterveld oppakken, dus hier het adres van het code-veld van X;
ii) het adres van de volgende cel in het parameterveld (dus 1012) op de terugkeer-stapel plaatsen;
iii) bewerkstelligen dat {X} wordt uitgevoerd.

Wanneer dit laatste is gebeurd, haalt de adres-interpretator de waarde 1012 van de terugkeer-stapel af om vervolgens het volgende adres, in dit voorbeeld voor de verwerking van het woord @, op te halen. Ook deze verwerking geschiedt volgens de bovenstaande drie stappen.

De laatste toevoeging aan het parameterveld van {Xkw}, het woord {EXIT}, is een gevolg van de afsluitende komma-punt. Het woord {EXIT} heeft bij verwerking het resultaat dat teruggekeerd wordt naar het volgende hogere niveau van verwerking (door een adres van de terugkeer-stapel af te halen en dit door te geven naar de adres-interpretator). Dit betekent dat wanneer {Xkw} opgenomen is in een andere colon-definitie, zoals:

```
: TEST Xkw X @ ;
```

bij verwerking van {TEST} tijdens de uitvoering van {Xkw} het adres van het volgende woord in {TEST}, dus van {X}, op de terugkeer-stapel staat. Wanneer dan {EXIT} voor {Xkw} is uitgevoerd, zal de adres-interpretator met {X} beginnen (dit is de volgende opdracht in TEST), hetgeen de bedoeling is. Dit gebruik van de terugkeer-stapel maakt het mogelijk om colon-definities te 'nesten'.

We kunnen in colon-definities (maar niet in DO-lussen!) het woord {EXIT} desnoods ook gebruiken om voortijdig naar het volgende hogere niveau van verwerking terug te keren. Daar dit het gestructureerde karakter van FORTH geweld aandoet, is dit gebruik van {EXIT} af te raden!

We kunnen {Xkw} ook direct intikken. Het zal worden uitgevoerd en er zal ok worden afgedrukt. We kunnen na deze ok weer een nieuwe opdracht intikken. Wat je je af kunt vragen is hoe {EXIT} dit voor elkaar krijgt. Hoe weet het systeem wat de volgende uit te voeren opdracht is? Het antwoord komt misschien als een verrassing, maar een FORTH systeem is altijd een colon-definitie aan het uitvoeren, zelfs wanneer op een invoersliert wordt gewacht! Door {EXIT} keren we terug naar de buitenste colon-definitie, die er in grote trekken als volgt uitziet (in feite dezelfde {QUIT} als in paragraaf 5.8):

```
: QUIT BEGIN
            ( Maak de terugkeer-stapel schoon )
            ( Lees met QUERY een invoersliert )
            ( Voer die uit met EXECUTE )
              " ok" CR 0 UNTIL ;   ( nooit eindigende lus )
```

Dit is de colon-definitie op het hoogste niveau. Aan de uitvoering hiervan is de computer altijd bezig ook al lijkt het alsof hij af en toe niets staat te doen.

9.6 Het definiëren van nieuwe vertaal-woorden

Nu we weten wat de structuur is van het FORTH-Woordenboek, kunnen we nieuwe 'vertaal-woorden' definiëren, maar eerst zullen we enkele woorden bespreken waarmee we het Woordenboek en het vertalen kunnen aanpakken.

De twee woorden {FIND} en {'} verschaffen ons inlichtingen over een bepaalde toevoeging aan het Woordenboek. Met {FIND} <naam> komt op de stapel het adres van het code-veld van de toevoeging voor <naam> (of een nul als die toevoeging ontbreekt). Zo vinden we met:

```
FIND Test . 12345 ok      ( adres van code-veld van Test )
```

dat {Test} aanwezig is in het Woordenboek. Het woord {EXECUTE} zal bewerkstelligen dat de opdracht waarvan het adres van het code-veld op de stapel staat wordt uitgevoerd, zodat:

```
FIND Test EXECUTE          en          Test
```

precies hetzelfde doen. Een meer bruikbare toepassing van {EXECUTE} is het 'indirect' uitvoeren van een opdracht, gebruik makend van het in een tabel opgeslagen adres van een code-veld. Een voorbeeld hiervan zullen we later in deze paragraaf tegenkomen.

Met {' <naam>} krijgen we op de stapel het adres van het parameter-veld van de toevoeging voor <naam> aan het Woordenboek (als die toevoeging er is). Zo kunnen we met:

```
10 CONSTANT a
1 ' a !
```

de waarde van een constante veranderen.

Wanneer {'} binnen een colon-definitie wordt gebruikt, heeft dit het effect dat het direct volgende woord als {naam} wordt opgevat (en niet het volgende woord in de invoersliert). Bijvoorbeeld:

```
: Testapv ' Test ; ok
Testapv . 12347 ok        ( adres van parameterveld van Test )
```

De 'voorrangs'-bit, vermeld in figuur 9.2 van de vorige paragraaf, bepaalt of een woord binnen een colon-definitie wordt vertaald of uitgevoerd. Woorden met een voorrangs-bit gelijk aan 1, worden 'onmiddellijke' woorden genoemd omdat ze onmiddellijk worden uitgevoerd wanneer de colon-definitie waar ze deel van uitmaken, vertaald wordt. Met het woord {IMMEDIATE} kunnen we van het laatst toegevoegde woord de voorrangsbit de waarde 1 geven. In bijvoorbeeld:

```
: Printnu CR ." Vertalend ..." CR ; IMMEDIATE ok
```

wordt het woord {Printnu} altijd uitgevoerd en wordt dus geen vertaalde tekst geproduceerd. In het volgende voorbeeld wordt hier weer gebruik van gemaakt.

```
: Test Printnu CR ." Verwerkend ..."  CR ;
Vertalend ...
ok
Test
Verwerkend ...
ok
```

Door het definiërende woord {:} schakelt FORTH om naar de vertaalfase, door het afsluitende woord {;} weer naar de uitvoeringsfase. Met de systeem-variabele {STATE} is vastgelegd in welke fase FORTH op een willekeurig moment is. Bijvoorbeeld:

```
: Fase? STATE @ IF ." Vertalend " ELSE ." Verwerkend " THEN ; ok
IMMEDIATE ok

Fase? Verwerkend ok                    ( Verwerkingsfase )
: Test 4 + Fase? ; Vertalend ok        ( Vertaalfase )
```

Omdat van het woord {Fase?} een onmiddellijk woord is gemaakt, produceert het geen vertaalde tekst.

De twee woorden {[}, resp. {]}, schakelen FORTH om naar de verwerkingsfase, resp. vertaalfase. Bijvoorbeeld:

```
: Test ." Print later " [ ." Print nu " ] ; Print nu ok
Test Print later ok
```

De door deze rechte haken ingesloten FORTH tekst wordt uitgevoerd tijdens de vertaling van {Test} en produceert geen vertaalde tekst. Het uitvoeren van berekeningen tijdens de vertaling is een zinvoller toepassing van deze omschakel-mogelijkheid. Bijvoorbeeld in:

```
1024 CONSTANT 1K ok
: Add_5K [ 1K 5 * ] LITERAL + ; ok
```

De laatste definitie is equivalent met:

```
: Add_5K 5120 + ; ok
```

Het woord {LITERAL} zet het met de tekst tussen rechte haken berekende resultaat op de top van de stapel (zo berekende getallen heten in FORTH 'literals'). Dus zijn:

```
: Tien [ 10 ] LITERAL ; ok    en    : Tien 10 ; ok
```

gelijkwaardige definities, die dezelfde vertaalde tekst produceren.

Opmerking. Literals produceren in feite twee dingen in het parameterveld. Het eerste is een verwijzing naar de verwerking van literals en het tweede het getal. Door de verwerking wordt dit getal op de stapel gezet.

Een interessante toepassing van] is het maken van een tabel van adressen van code-velden. Het woord {EXECUTE} kan vervolgens worden gebruikt om een van de bij de code-velden behorende woorden uit te voeren. Bijvoorbeeld:

```
: NUL   ." nul  " ; ok                ( drie woorden )
: EEN   ." een  " ; ok
: TWEE  ." twee " ; ok

CREATE tabel ] NUL EEN TWEE [ ok      ( maak tabel )
: Kies 2 * tabel + @ EXECUTE ; ok

0 Kies nul ok
2 Kies twee ok
```

Het woord {COMPILE} kan worden gebruikt om nieuwe woorden te definiëren, die zowel een verwerking als een vertaling kunnen bewerkstelligen. De definitie van zo'n 'compilatie' - woord heeft de volgende structuur:

```
: verwerkingsactie ..... ;
: compilatie-woord COMPILE verwerkingsactie
                   .. compilatie opdrachten .. ; IMMEDIATE
```

Aangezien het (nieuwe) compilatie-woord een onmiddellijk woord is, wordt het uitgevoerd wanneer het in een colon-definitie voorkomt. Het woord {COMPILE} zal dan het adres van het code-veld voor de eerder gedefinieerde {verwerkingsactie} maken, en de {compilatie opdrachten} worden meteen uitgevoerd.

Het woord {LITERAL} is een voorbeeld van zo'n compilatie-woord daar het als volgt gedefinieerd kan worden:

```
: (LITERAL)  R>          ( adres van getal )
             DUP         ( dupliceer het )
             2+ >R       ( wijst naar volgende cel in code-veld )
             @ ;         ( zet getal op de stapel )

: LITERAL    COMPILE (LITERAL)
             ,           ( doe getal in het Woordenboek )
             ; IMMEDIATE
```

Tijdens de vertaling wordt het getal op de top van de stapel door {,} in de volgende cel in het Woordenboek opgeborgen. Wanneer tijdens de verwerking {(LITERAL)} wordt uitgevoerd, wijst de waarde op de top van de terugkeer-stapel naar de volgende cel in het code-veld, waarin het getal bewaard is. Dit getal komt dan op de (gewone) stapel en het adres op de terugkeer-stapel wordt met twee verhoogd, zodat de adres-interpretator de cel met het getal verder overslaat.

De 'DO'- en 'IF'-structuren zijn ook voorbeelden van compilatie-woorden die tijdens de verwerking het adres op de terugkeer-stapel veranderen

om de adres-interpretator te dwingen het programma op een andere plaats te vervolgen. Ter toelichting volgt nu de definitie van een nieuwe lusstructuur, de 'STEP...DOWN':

```
: STEP COMPILE >R              ( lus-teller op de terugkeer-stapel )
       HERE ; IMMEDIATE        ( zet HERE op de terugkeer-stapel )

: (DOWN) R>                    ( pak terugkeer-adres )
         R>                    ( en de lus-teller )
         -1 DUP                ( verlaag lus-teller )
           IF >R               ( bewaar nieuwe teller-waarde )
              @  >R            ( en sprongadres )
           ELSE                ( einde van de lus )
               DROP            ( gooi teller-waarde weg )
               2+ >R           ( en spring over sprongadres )
           THEN ;

: DOWN  COMPILE (DOWN)         ( maak verwerkingsprogramma )
        , ; IMMEDIATE          ( bewaar HERE uit STEP in Woorden-
                                                        boek )
```

Tijdens de vertaling zet {STEP} het met {HERE} verkregen huidige vrije ruimte adres in het Woordenboek op de stapel, en de corresponderende {DOWN} plaatst dit adres met {,} in het code-veld. Het verwerkingsprogramma {(DOWN)} zet dit sprongadres met {@ >R} op de terugkeer-stapel wanneer de lus herhaald moet worden, of zorgt met {2+ >R} voor een sprong er over heen na afloop van de lus. De lus-teller wordt op de terugkeer-stapel bewaard met {>R} tijdens uitvoering van {STEP}. Een voorbeeld van het gebruik van deze nieuwe lus-constructie is:

```
: Test 10 STEP I . DOWN ; ok
Test 10 9 8 7 6 5 4 3 2 1 ok
```

Tenslotte moet het woord {[COMPILE]} worden vermeld. Dit wordt benut om de voorrangs-bit in onmiddellijke woorden te niet te doen, zodat ze in colon-definities kunnen worden opgenomen en uitgevoerd tijdens de verwerkingsfase en niet tijdens de vertaalfase. Voor een voorbeeld van gebruik van dit woord nemen we aan dat we het volgende woord in een invoersliert willen 'afvinken'. Dit kan met:

```
: .pva [COMPILE] ' . ; ok      ( druk adres parameterveld af )
.pva Test 12347 ok
```

9.7 Samenvatting

In dit hoofdstuk zijn de volgende nieuwe woorden ingevoerd:

Definiërende woorden

DOES> (→)

Definieert het begin van de verwerkingsactie van een nieuw definiërend woord in de constructie:

```
: defwoord ..... CREATE ... DOES>  .... ;
```

Wordt met:

```
defwoord <naam>
```

het woord 'naam' ingevoerd, dan worden bij uitvoering van 'naam' de woorden tussen DOES> en de komma-punt uitgevoerd met op de stapel het adres van het parameterveld van 'naam'.

Woordenboekbeheer

' (→ adres)

Door ' <naam> wordt het adres van het parameterveld voor de toevoeging van <naam> aan het Woordenboek op de stapel gezet. Is dit gebruikt binnen een colon-definitie, dan is dit adres als een literal in het Woordenboek gezet. Wanneer <naam> niet in het Woordenboek is te vinden, komt er een foutmelding.

FIND (→ adres)

Zet het adres van het code-veld van het volgende woord in de invoersliert op de stapel, of anders een nul als dit woord niet te vinden is in het Woordenboek.

Vertaal-woorden en besturingsstructuren

IMMEDIATE (→)

Kenmerk de laatste toevoeging aan het Woordenboek als een uit te voeren woord, ook al staat het in een colon-definitie.

LITERAL (n →)

Zet in de vertaalfase n als een 16 bits literal in het Woordenboek. Hierdoor zal bij latere uitvoering n op de stapel blijven staan

STATE (→ adres)

Een systeem-variabele die aangeeft of het systeem aan het vertalen of aan de uitvoering bezig is. Een niet-nul waarde geeft de vertaalfase aan.

[(→)

Einde van de vertaalfase, zodat de tekst daarna verwerkt wordt.

] (→)

Overgang naar vertaalfase voor de daarop volgende tekst.

COMPILE (→)

Wanneer een woord met {COMPILE} in de definitie wordt uitgevoerd, komt het adres van het code-veld van het woord na {COMPILE} in het Woordenboek.

[COMPILE] (→)

Het woord hierop volgend wordt vertaald, zelfs als het een onmiddellijk woord is.

EXECUTE (n →)

Voer de opdracht uit, waarvan het adres van het code-veld op de stapel staat.

EXIT (→)

Als {EXIT} binnen een colon-definitie staat doet {EXIT} hetzelfde als {;}. Mag niet binnen een DO-lus worden gebruikt.

10 FORTH FINALE

In dit laatste hoofdstuk zullen we het tot dusver geleerde in twee complete FORTH programma's in de praktijk brengen. In hoofdlijnen zullen we de ontwikkelingsfasen in ieder programma aangeven, dus het beginontwerp, gevolgd door de uitwerking en beproeving en tenslotte de uiteindelijke woordenschat. Eerst komt de kalender woordenschat, die in de inleiding genoemd werd, en daarna een interactief beeldschermspel. Deze voorbeelden zijn gekozen omdat de programma's zelf interessant zijn en omdat ze ieder een bepaald type programmeringsprobleem belichten. Het kalenderprogramma bevat nogal wat rekenwerk en het beeldschermspel berust op het snel tekenen van figuren op het beeldscherm. Geen van de programma's vereist het gebruik van schijven, zodat de programma's ook gedraaid kunnen worden op een systeem met cassetteband.

10.1 Een kalender woordenschat

Een bruikbare verzameling 'kalender woorden' is al voorgesteld in de inleiding van dit boek. Deze woorden zijn:

dag	(dag maand jaar →)	Druk de naam van de dag ('??dag') voor de gegeven datum af.
maand	(maand jaar →)	Druk een kalenderblad af voor de opgegeven maand.
jaar	(jaar →)	Druk een kalender af voor het opgegeven jaar.
resterend	(dag maand jaar →)	Druk voor de opgegeven datum het resterende aantal dagen in een jaar af.

Na deze specificatie van het gewenste eindresultaat moeten we voor het bereiken daarvan een aanpak ontwikkelen, m.a.w. een aantal verstandige deelresultaten specificeren waarmee we het eindresultaat kunnen opbouwen. De voornaamste bewerking in een kalender programma is het bepalen van de '??dag' waarop de eerste januari van een jaar valt. Gelukkig bestaat daarvoor de formule van Zeller, die we zullen gebruiken om het woord {1jan} te definiëren. Vanzelfsprekend is de volgende benodigde bewerking de berekening van het dagnummer voor de gegeven datum met behulp van het woord {dagn}. Door combinatie van {1jan} en {dagn} is het eerste woord van onze kalender woordenschat dan eenvoudig te definiëren.

10.1.1 De formule van Zeller

Blokken 100 en 101 (zie 10.1.4 voor de volledige programma's).

De volgende, wat ingewikkeld lijkende, formule geeft (voor de Gregoriaanse kalender, dus vanaf 1582) voor een willekeurig jaar de dag D in de week waarop de eerste januari valt. De dagen van zondag tot zaterdag zijn daarbij van 0 tot 6 genummerd. Voorts gebruiken we de notatie int(a) om het gehele getal aan te geven dat uit a ontstaat door weglaten in a van de cijfers achter de komma, dus int(0.8) = 0. De dag D in het jaar j vinden we dan met (A en B zijn hulpvariabelen):

```
A = int((j-1)/100)
B = j-1-100*A
D = 799+B+int(B/4)+int(A/4)-2*A
D = D MOD 7
```

Dit is ook direct in FORTH op te schrijven:

```
VARIABLE j  ( jaar )
VARIABLE A  ( hulpvariabele )
VARIABLE B  ( hulpvariabele )
: 1jan j @ 1 - 100 /  A !
       j @ 1 - 100 A @ * - B !
       799 B @ + B @ 4 / + A @ 4 / + 2 A @ * -
       7 MOD ;
```

Dit zou ook zonder de hulpvariabelen opgeschreven kunnen worden door de stapel te gebruiken voor het bewaren van tussenresultaten tot ze nodig zijn. De meeste programmeurs zullen het er over eens zijn dat dit vanwege de extra moeilijkheden bij het schrijven en testen niet verantwoord zou zijn, temeer omdat deze berekening niet veel tijd vergt en waarschijnlijk maar een keer hoeft te worden gedaan in een kalenderberekening.

Door het bovenstaande in een nieuw blok op schijf of cassette op te nemen en het daarna te 'LOADen', kunnen we het programma als volgt testen:

```
1982 j ! ok
1jan . 5 ok
```

Aangezien het resultaat 5 volgens afspraak 'vrijdag' voorstelt is gemakkelijk met een zakagenda te controleren dat 1 januari 1982 inderdaad op een vrijdag viel. We controleren ook nog dat {1jan} geen ongewenste getallen achterlaat op de stapel met:

```
. 0 STACK EMPTY
```

Op dit moment realiseren we ons ook dat we eigenlijk ook een woord nodig hebben dat een '??dag' afdrukt met "zondag", "maandag", enz. omdat dit mooier is en het testen van {1jan} vereenvoudigt. Daarom definiëren we:

```
: "dagen" ." zondag    " ." maandag   "  ." dinsdag  "
          ." woensdag " ." donderdag"  ." vrijdag  "
          ." zaterdag " ;
: printdag 12 * ' "dagan" + 3 + 9 TYPE ;
```

Het woord {"dagen"} bevat een lijst met strings van gelijke lengte. Iedere string beslaat precies 12 bytes van het parameterveld; de eerste twee bytes bevatten het adres voor {."}, de derde byte is voor het aantal bytes en de laatste negen bytes bevatten de '??dag'. Het woord {printdag} vermenigvuldigt het getal op de stapel (dat tussen 0 en 6 moet liggen) met 12 en telt er het met {"dagen"} verkregen adres van het parameterveld bij op. Daarbij wordt nog 3 opgeteld om het beginadres van de '??dag' te krijgen, die dan met {9 TYPE} wordt afgedrukt.

Een eenvoudige 'diagnostiek' definitie stelt ons nu in staat om zowel {1jan} als {printdag} grondig te testen:

```
: test1 1985 1980 DO
                    I j !
                    ." 1 jan " I . SPACE
                    1jan
                    ." - " printdag CR
                  LOOP
```

omdat dit het volgende moet opleveren:

1 jan 1980 - dinsdag
1 jan 1981 - donderdag
1 jan 1982 - vrijdag
1 jan 1983 - zaterdag
1 jan 1984 - zondag

Deze techniek van tussendoor-diagnostiek-definities tijdens het ontwikkelen van de hoofddefinities is zeker de moeite waard en helpt bij het testen!

10.1.2 Dagnummer en dag

Blokken 102 en 103

Het woord {dagn} moet voor een gegeven datum het aantal dagen vanaf het begin van het jaar berekenen, zodat bijvoorbeeld 2 februari de 33-ste dag is (31 dagen in januari + 2). De beste aanpak is om eerst een array van constanten, het aantal dagen in iedere maand voorstellend, te maken met de in paragraaf 3.5 beschreven methode. Daar we alleen met kleine getallen te maken hebben, is een byte array aangewezen:

```
CREATE dpmtabel          ( dagen per maand )
     31 C, 28 C, 31 C, 30 C, 31 C, 30 C,
     31 C, 31 C, 30 C, 31 C, 30 C, 31 C,
: dpm  dpmtabel + C@ ;
```

Het is nu eenvoudig {dagn} op te schrijven:

```
VARIABLE d                              ( dag )
VARIABLE m                              ( maand )
: dagn 0                                ( beginwaarde )
       12 0 DO
              m @ I =
                IF   d @ + LEAVE
                ELSE I dpm +
                THEN
            LOOP ;
```

Voor de gegeven dag d en maand m geeft {dagn} dan het gezochte aantal dagen met een lus voor de maanden januari (0) t/m december (11). Is de opgegeven maand m bereikt, dan wordt de dag d opgeteld bij het tot dusver op de stapel verkregen {dagn} en de lus wordt met {LEAVE} verlaten. Is m nog niet bereikt, dan wordt met behulp van {dpm} het aantal dagen van de maand bij {dagn} opgeteld en de lusopdrachten worden herhaald.

Het eerste gezochte kalenderwoord is nu direct op te schrijven:

```
: dag j ! m ! d !
      1jan dagn + 1- 7 MOD
      printdag ;
```

De aftrekking van 1 hangt samen met de nummering van de dagen terwijl {7 MOD} het gezochte nummer van de dag geeft en {printdag} voor een nette afdruk zorgt. Bijvoorbeeld:

```
31 11 1981 dag donderdag ok          ( 31 december 1981 )
1 0 1982 dag vrijdag ok              ( 1 januari 1982 )
```

Wat in het voorgaande echter nog niet goed is, is dat in {dpm} en dus evenzo in {dagn} geen rekening is gehouden met schrikkeljaren. Ook wordt in {dag} niet gecontroleerd of de opgegeven datum bestaat of zal bestaan. De twee hiervoor benodigde correcties zijn, evenals een verfraaiing voor de maandnamen, vrij direct op te schrijven, zoals te zien is in de volledige programmatekst in paragraaf 10.1.4.

10.1.3 Maand, jaar en 'resterend'

Blokken 104 en 105.

In de definitie van {maand} is de grootste moeilijkheid te zorgen voor een nette afdruk, waarin de dagnummers netjes in de goede kolommen staan, zoals bijvoorbeeld:

```
 Z    M    D    W    D    V    Z
                             1    2
 3    4    5    6    7    8    9
10   11   enz.
```

We moeten daartoe weer de dag bepalen waarop de eerste van de maand valt, en vervolgens spaties afdrukken tot de kolom van die '??dag'. Daarna drukken we achtereenvolgens de nummers van de dagen af, er voor zorgend dat na iedere 'zaterdag-kolom' op een nieuwe regel wordt begonnen. Om bij te houden in welke kolom we zitten, gebruiken we een 'karakter-teller', die opgehoogd wordt wanneer getallen of spaties zijn afgedrukt.

Op deze manier is de volgende definitie tot stand gekomen:

```
0    VARIABLE kart
1    : maand j ! m ! 1 d !
2           SPACE ."   Z   M   D   W   D   V   Z"
3           1jan dagn + 1- 7 MOD
4           4 * DUP SPACES kart !
5           m @ dpm 1+ 1 DO
6                        I 4 .R  4 kart +!
7                        kart @ 24 > IF CR 0 kart ! THEN
8                      LOOP ;
```

De opdrachten in regel 3 zijn dezelfde als die in {dag} en leveren de '??dag' voor de eerste van de maand omdat de variabele d de waarde 1 heeft gekregen (in regel 1). Die '??dag' wordt in regel 4 gebruikt om het benodigde aantal spaties, ook vastgelegd in {kart}, af te drukken. In regel 5 begint de DO-lus voor het aftellen van de dagen van de maand {m}. Door {I 4 .R} op regel 6 wordt een getal rechts in zijn kolom afgedrukt (als je systeem niet over {.R} beschikt, is een definitie daarvan in paragraaf 8.4 te vinden). Met {4 kart +!} wordt de karakter-teller opgehoogd, waarvan de waarde op regel 7 wordt bekeken voor een mogelijke {CR}.

De {maand}-versie in blok 104 van de volgende paragraaf laat nog enkele verfijningen zien, zoals een datum-controle met {datumc} en het afdrukken van de naam van de maand. Hiermee krijgen we bijvoorbeeld:

```
februari 1982 maand
februari   : 1982
  Z   M   D   W   D   V   Z
      1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28

ok
```

De definitie van {jaar} is nu direct op te schrijven, zoals blok 104 laat zien. Het laatste woord in onze kalender woordenschat, {resterend}, vertoont maar één lastig probleem wanneer we onder het einde van het jaar, {jaareind}, een willekeurige datum verstaan zoals het geval is bij verenigingsjaar of maatschappijjaar. Dit einde van het jaar kan in het volgende kalenderjaar (mogelijk een schrikkeljaar!) liggen.

Het woord {dagn}, zoals gedefinieerd in blok 103, houdt wel rekening met schrikkeljaren, maar om het resterende aantal dagen in het lopende jaar te berekenen, moeten we 'dagn' aftrekken van hetzij 365, hetzij 366. De definitie van {resterend}, zoals in blok 105 vermeld, wordt als volgt gebruikt:

```
1 juni jaareind ok                ( alleen dag en maand opgeven )
1 maart 1982 resterend 92 ok
2 juni 1982 resterend 364 ok
```

10.1.4 De blokken voor de kalender woordenschat

```
100 LIST
 0 ( Kalender woordenschat; formule van Zeller )
 1 DECIMAL
 2 FORTH DEFINITIONS
 3 VOCABULARY kalender
 4 kalender DEFINITIONS
 5
 6 VARIABLE j  VARIABLE m  VARIABLE d     ( jaar, maand, dag )
 7
 8 VARIABLE A  VARIABLE B      ( hulpvariabelen voor 1jan )
 9 : 1jan       ( levert de dag, 0-6, voor 1 januari in jaar j )
10       j @ 1 - 100 / A !
11       j @ 1 - 100 A @ * - B !
12       799 B @ + B @ 4 /+ A @ 4 / + 2 A @ * -
13       7 MOD ;                                  ( → n)
14
15 101 LOAD 102 LOAD 103 LOAD 104 LOAD 105 LOAD

101 LIST
 0 ( Kalender woordenschat; afdrukken van strings )
 1 : "dagen"    ( string-tabel voor dagen van de week )
 2    ." zondag   " ." maandag  " ." dinsdag  " ." woensdag "
 3    ." donderdag" ." vrijdag  " ." zaterdag " ;
 4 : printdag   ( druk de naam van de dag af )
 5    12 * ' "dagen" + 3 + 9 TYPE ;              ( n → )
 6
 7 : "maanden"  ( string-tabel voor namen van de maanden )
 8    ." januari  " ." februari " ." maart    " ." april    "
 9    ." mei      " ." juni     " ." juli     " ." augustus "
10    ." september" ." oktober  " ." november " ." december "
11 : printmaand ( druk de naam van de maand af )
12    12 * ' "maanden" + 3 + 9 TYPE ;            ( n → )
13
14
15
```

```
102 LIST
 0 ( Kalender woordenschat; controle van gegevens )
 1 CREATE dpmtabel   ( tabel van dagen per maand )
 2     31 C, 28 C, 31 C, 30 C, 31 C, 30 C,
 3     31 C, 31 C, 30 C, 31 C, 30 C, 31 C,
 4 : schr? j @ 4   MOD 0=      ( is jaar j een schrikkeljaar? )
 5         j @ 100 MOD 0= NOT AND
 6         j @ 400 MOD 0= OR ;                        ( → b)
 7 : dpm   DUP dpmtabel + C@
 8         SWAP 1 = schr? AND  ( tel 1 op voor februari in )
 9         IF 1+ THEN ;                  ( een schrikkeljaar )
10 (controleer of datum bestaat of zal bestaan )
11 : jcont j @ DUP 1582 < SWAP 4902 > OR ;          ( → b)
12 : mcont m @ 12 U< NOT ;                          ( → b)
13 : dcont d @ 1 - m @ dpm U< NOT ;                 ( → b)
14 : datumc jcont mcont dcont OR OR
15          IF ." Geen goede datum" ABORT THEN ;

103 LIST
 0 ( Kalender woordenschat; dagnummer en dag ) : C CONSTANT ;
 1 0 C januari    1 C februari   2 C maart      3 C april
 2 4 C mei        5 C juni       6 C juli       7 C augustus
 3 8 C september  9 C oktober   10 C november  11 C december
 4
 5 dagn 0 12 0 DO          ( bereken dagen tot d/m/j )
 6              m @ I = IF    ( lus voor maanden )
 7                       d @ + LEAVE   ( tot m=I )
 8                     ELSE
 9                       I dpm +
10                     THEN
11            LOOP ;                        ( → n )
12 ( bereken dag in de week voor datum d/m/j, 0-6 )
13 : d/m/j 1jan dagn + 1- 7 MOD ;           ( → n )
14 : dag j ! m ! d ! datumc     ( druk gezochte dag af )
15       d/m/j printdag ;                   ( d m j → )

104 LIST
 0 ( Kalender woordenschat; maand en jaar )
 1 VARIABLE kart                            ( karakter-teller )
 2 : maand j ! m !  1 d ! datumc     ( druk maand af )
 3         CR m @ printmaand SPACE ." :" j @ .  ( kopregel )
 4         CR SPACE ."   Z   M   D   W   D   V   Z" CR
 5         d/m/j          ( bereken eerste dag van de maand )
 6         4 * DUP SPACES kart !     ( ga naar goede kolom )
 7         m @ dpm 1+ 1 DO           ( dagen afdrukken )
 8                       I 4 .R  4 kart +!
 9                       kart @ 24 > IF CR 0 kart ! THEN
10                     LOOP CR CR ;      ( m j → )
11
```

```
12 : jaar                ( druk een kalender voor een jaar af )
13      12 0 DO          ( lus voor alle maanden )
14            I OVER maand
15          LOOP DROP ;                          ( j →   )

105 LIST
 0 ( Kalender woordenschat; jaareind en resterend )
 1 VARIABLE mend VARIABLE dend      ( einde van het jaar )
 2 : jaareind
 3         OVER OVER 1 = SWAP 29 = AND       ( 29 februari ? )
 4         IF ." Dat is een grapje!" ABORT THEN
 5         mend ! dend ! ;                    ( d m →   )
 6 : dinj                ( hoeveel dagen telt jaar j ? )
 7         schr? IF 366 ELSE 365 THEN ;      (   → n )
 8 resterend
 9         j ! m ! d ! datumc dagn
10         mend @ m !  dend @ d ! datumc dagn
11         OVER OVER > NOT IF       ( datum voor jaareind ? )
12                           SWAP - .
13                        ELSE dinj SWAP -
14                             1 j +! datumc dagn + .
15                        THEN ;             ( d m j →   )
```

10.2 Een woordenschat voor een beeldschermspel

Het hier uitgewerkte spel is 'solo squash' (van een speler tegen de computer). De speler kan daarbij een 'bat' van links naar rechts over de onderste regel van een beeldscherm bewegen. De computer 'serveert' vanaf de bovenste regel van het scherm in een willekeurige richting een bal naar beneden met een in te stellen snelheid. Wordt met het bat de bal onderschept, dan kaatst die terug naar boven en de speler verdient enkele punten. Bij missen van de bal verliest de speler punten en wordt opnieuw een bal geserveerd. Wanneer de bal de bovenkant of de zijkanten van het scherm raakt, of het midden van het bat, dan is de botsing volkomen elastisch. Wordt het bat niet in het midden geraakt, dan kaatst de bal daarentegen in een willekeurige richting terug.

Door de grote verwerkingssnelheid is FORTH bijzonder geschikt voor spelletjes van dit type. Eigenlijk moet die snelheid meestal nog verlaagd worden om zo'n spel ook voor iemand zonder bliksemsnelle reactie aantrekkelijk te maken. Interessant is ook dat veel programma's voor de spelletjes in een speelhal tegenwoordig vaak in FORTH zijn geschreven.

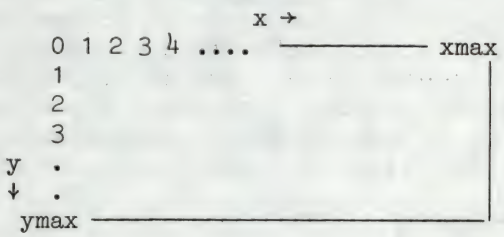
Voor een doeltreffende structurering van de benodigde woordenschat is het raadzaam om een opzet voor het uiteindelijke woord, {squash}, meteen in het begin te schetsen, zodat we daardoor kunnen voorzien welke andere woorden nodig zullen zijn. De eenvoudigste manier om dit te doen is het opschrijven van commentaar, zoals op de volgende pagina's is te zien. Daarna moet het commentaar worden vervangen door definities.

```
: squash
    ( veeg scherm schoon en zet score=0 op scherm )
    ( serveer een bal )
    BEGIN
       ( zet bal en bat op het scherm )
       ( raadpleeg toetsenbord en verplaats eventueel bat )
       ( verplaats de bal, eventueel na terugkaatsing )
       IF          ( bal is op de onderste regel )
         IF        (   en bat is geraakt  )
           ( kaats de bal weer terug )
           ( verhoog de score )
         ELSE      ( bat mist de bal )
           ( verlaag de score )
           ( serveer opnieuw een bal )
         THEN
         ( zet nieuwe score op scherm )
       THEN
  0 UNTIL ;        ( opnieuw de lus in )
```

10.2.1 Behandeling van de bal

Blokken 110 en 111 (zie 10.2.4 voor volledige programma's)

Bij ieder beeldschermspel moet je op een willekeurige plaats op het scherm iets kunnen veranderen zonder het hele beeld te moeten vernieuwen. Dit kan inderdaad met een beeldscherm-buffer, waarin ieder beeldpunt ('pixel') direct geadresseerd en veranderd kan worden. Figuur 10.1 illustreert de hier te gebruiken adressering van zo'n beeldscherm. Het beeldpunt kan een puntje zijn bij een fijne schermverdeling of een blokje van puntjes bij een grove verdeling.



Figuur 10.1 Afbeelding van het beeldscherm

Bij het adresseren van het scherm beginnen we links-boven en eindigen rechts-beneden. Het plotten van een beeldpunt op de plaats (x,y) kan dan bijvoorbeeld als volgt gebeuren:

```
HEX   F000 CONSTANT vdust      ( begin van schermbuffer )
DECIMAL 64 CONSTANT breedte    ( 64 karakters schermbreedte )
        16 CONSTANT hoogte     ( 16 regels schermhoogte )
: coord breedte * + vdust + ; ( x y → adres)
: plot coord C! ;              ( kar x y → )
```

In dit voorbeeld is {plot} gedefinieerd voor een grove schermverdeling van 16 regels van ieder 64 karakters. Een 'bal' met het symboolnummer 192 kan dan op de plaats 5,5 worden geplot met:

```
192 5 5 plot
```

Bij systemen met een fijne schermverdeling kun je bijvoorbeeld een bepaald geadresseerd beeldpunt laten oplichten. Daar voor deze systemen nog geen normen bestaan, zal hier echter worden aangenomen dat we met een grove schermverdeling moeten werken.

De behandeling van de bal zal gebeuren met enkele colon-definities, waarbij de momentane bal-positie zal worden vastgelegd met de coordinaten {x} en {y}, die echter niet buiten de grenzen 0 tot xmax en 0 tot ymax mogen komen. Dit wordt dan ook gecontroleerd in de volgende definities:

```
breedte 1- CONSTANT xmax     ( vastlegging van xmax en ymax )
hoogte 1- CONSTANT ymax
VARIABLE x VARIABLE y        ( bal-coordinaten )
: xycheck x @ 0    MAX x !   ( x nooit kleiner dan 0 )
          x @ xmax MIN x !   ( x nooit groter dan xmax )
          y @ 0    MAX y !   ( y nooit kleiner dan 0 )
          y @ ymax MIN y !   ( y nooit groter dan ymax )
: xyplot xycheck x @ y @ plot ;     ( kar → )
```

Zodra {x} of {y} buiten hun bereik dreigen te komen, zal {xycheck} ze weer terugdringen op hun grenzen. Merk op dat met {MAX} en {MIN} gecompliceerde IF-structuren worden voorkomen.

Het verplaatsen van de bal gebeurt met het woord {xystep}, waarin de waarden van de variabelen {xstep} en {ystep} opgeteld worden bij resp. {x} en {y}. Terugkaatsing van een kant gebeurt eenvoudig door tekenomkeer van {xstep} of {ystep}, afhankelijk van de kant waar een botsing plaats vindt. Met de definities {xlinks}, {xrechts}, {yboven} en {yonder} wordt nagegaan of de bal de linker-, rechter-, boven- of onder-kant heeft geraakt en wordt zo nodig de bal teruggekaatst. Met de test-definitie {patroon} in blok 111 worden deze opdrachten getest.

Figuur 10.2 Een test-patroon

Door {n patroon}, voor het plotten van n punten, in te tikken, krijgen we een figuur als 10.2 waarin de beweging van een bal aangegeven is voor bepaalde waarden van de verschillende parameters.

10.2.2 Behandeling van het bat

Blok 112

Een drie-beeldpunten-breed bat op de onderste regel van het beeldscherm krijgen we op de posities (31.15), (32.15) en (33.15) met:

```
32 bat !
163 plotbat
```

Het bat wordt een positie naar links verplaatst met {bat-1} en naar rechts met {bat+1}, waarbij tevens wordt gecontroleerd dat het bat niet van het scherm afgaat.

Door het woord {hupbat} wordt op de top van de stapel een van het toetsenbord afkomstig ASCII-karakter verwacht; is het de code voor "z" dan gaat het bat naar links, is het een "/" dan gaat het bat naar rechts, en is het iets anders dan wordt het spelletje gestopt. In het woord {zetbat} staan alle woorden voor de behandeling van het bat bij elkaar:

```
32 CONSTANT "blank"                ( spatie )
163 CONSTANT "bat"                 ( karakter voor bat )
: zetbat
     ?TERMINAL                     ( toets ingedrukt ? )
     ?DUP                          ( zo ja, dupliceer )
     IF
       "blank" plotbat             ( oude bat wegvegen )
       hupbat                      ( beweeg bat links of rechts )
       "bat" plotbat               ( plot nieuwe bat-positie )
     THEN ;
```

Het woord {?TERMINAL} is wel geen FORTH-79 standaard woord, maar de meeste systemen kennen het toch wel. Dit woord gaat na of een toets is ingedrukt en zet de corresponderende ASCII-code op de stapel als dat zo is, of anders een nul. Het woord {zetbat} doet dus niets als er geen toets is ingedrukt.

De volgende 'test-definitie' is goed te gebruiken:

```
: testbat  clrs                ( veeg scherm schoon )
           "bat" plotbat       ( plot het bat )
           BEGIN zetbat 0 UNTIL ;       ( verplaats het of stop )
```

Door {testbat} in te tikken wordt het scherm schoongeveegd en een bat geplot, dat dan met de "z" en "/" toetsen heen en weer verplaatst kan worden. Door een willekeurige andere toets in te drukken, wordt het programma beeindigd. Als je toetsenbord een 'repeat'-toets heeft, probeer het dan ingedrukt te houden terwijl je "z" of "/" aanraakt.

10.2.3 Het squash-spel

Blokken 113 en 114

De inleidende definities in blok 113 betreffen meestal het in een willekeurige richting terugkaatsen of bewegen van de bal. Zij maken daartoe gebruik van een bekende methode voor het produceren van een rij aselecte getallen ('random numbers'), die als volgt is geprogrammeerd:

```
VARIABLE rnd
1234 rnd !                ( beginwaarde van de rij )
: random rnd @            ( neem beginwaarde )
         1021 * 41 +      ( 1021 * rnd + 41 )
         DUP rnd ! ;      ( bewaar rnd )
```

Iedere keer als {random} wordt uitgevoerd, komt een nieuw aselect getal in het bereik van -32768 tot 32767 (behalve 0) op de stapel. Deze rij van pseudo-willekeurige getallen herhaalt zich na 65535 getallen, maar dat geeft hier geen problemen. Meestal willen we in een kleiner bereik aselecte getallen krijgen, wat mogelijk is met:

```
: >rand random U* SWAP DROP ;
```

Met {n >rand} krijgen we een getal in het bereik van 0 tot n-1.

Met het woord {nwbal} kiezen we voor {x} een waarde tussen 0 en 63 en {y} wordt 0, zodat een nieuwe bal ergens op de bovenste regel van het scherm begint. Voorts krijgt {xstep} een van de waarden 2, 1, -1 of -2 en wordt {ystep} 1 of 2, waardoor de bal in een willekeurige richting naar beneden gaat.

Met {wijkaf} krijgen alleen {xstep} en {ystep} een waarde, zodat een bal die het bat niet precies in het midden raakt in een willekeurige richting naar boven teruggekaatst wordt.

De overige woorden in blok 113 zijn voorbereidingen om het woord {squash} gemakkelijk te kunnen definiëren. De woorden {hitmid} en {hitlr} geven de waarde 'true' wanneer de bal het bat precies in het midden, resp. aan een van de kanten treft. Met {printscore} wordt op het scherm de laatste stand gegeven op de plaats van de oude.

Na al deze voorbereidingen is de definitie van {squash} vrijwel direct op te schrijven door de algoritme uit 10.2 te volgen. In het volgende zal men slechts één 'verbetering' aantreffen, nl. dat {zetbat} in een lus zit, die {vlug} keer wordt herhaald. Dit heeft twee effecten, in de eerste plaats dat de balbeweging zodanig wordt vertraagd dat tegenspel mogelijk wordt, en ten tweede dat voor iedere bal-positie het toetsenbord {vlug} keer wordt geraadpleegd voor een verplaatsing van het bat met behulp van de "z" en "/" toetsen. Met een 'vlug'-waarde van 400 is het een eenvoudig spel, met een waarde van 300 vrij moeilijk en met een waarde van 200 zeer enerverend. Probeer daarom met een zo klein mogelijke waarde van 'vlug' toch te winnen!

```
: squash
     clrs                              ( veeg scherm schoon )
     0 score ! printscore              ( score=0 op scherm )
     nwbal                             ( serveer een bal )
     BEGIN
       "bal" xyplot                    ( zet bal op scherm )
       "bat" plotbat                   ( zet bat op scherm )
       vlug @ DO
                zetbat                 ( bat verplaatsen ? )
              LOOP
       "blank" xyplot                  ( bal wegvegen )
       xystep                          ( bal verplaatsen )
       xlinks xrechts yboven           ( boven- of zij-kanten ? )
       y @ ymax >                      ( bal niet boven onderkant? )
            IF hitmid                  ( bat middenop geraakt ? )
                 IF 10 score +!        ( 10 punten er bij )
                 ykeer                 ( bal teruggekaatst )
                 ELSE hitlr            ( bat aan kant geraakt ? )
                    IF 5 score +!      ( 5 punten er bij )
                       wijkaf          ( bal scheef terug )
                    ELSE -5 score +!   ( 5 punten er af )
                         nwbal         ( serveer nieuwe bal )
                    THEN
                 THEN
                 printscore            ( nieuwe score op scherm )
            THEN                       ( opnieuw de lus in )
   0 UNTIL ;
```

10.2.4 De blokken voor het beeldschermspel

```
110 LIST
 0 ( Woordenschat voor beeldschermspel; plot-programma's )
 1 FORTH DEFINITIONS            ( voor een nieuwe woordenschat )
 2 VOCABULARY beeldschermspel  beeldschermspel DEFINITIONS
 3 HEX   F000 CONSTANT vdust   ( systeem-constanten )
 4 DECIMAL 80 CONSTANT breedte 25 CONSTANT hoogte
 5 : coord breedte * + vdust + ;          ( x y → adres )
 6 : plot  coord C! ;                     ( kar x y  →  )
 7 breedte 1- CONSTANT xmax  hoogte 1- CONSTANT ymax
 8 VARIABLE x VARIABLE y       ( bal-coordinaten )
 9 : xycheck x @    0 MAX x !  ( x nooit kleiner dan 0 )
10          x @ xmax MIN x !  ( x nooit groter dan xmax )
11          y @    0 MAX y !  ( y nooit kleiner dan 0 )
12          y @ ymax MIN y !  ( y nooit groter dan ymax )
13       ( plot karakter op positie x,y met controle )
14 : xyplot xycheck x @ y @  plot ;      ( kar  →     )
15 111 LOAD 112 LOAD 113 LOAD 114 LOAD   ( laad de rest )
```

```
111 LIST
 0 ( Woordenschat voor beeldschermspel; bal behandeling )
 1 VARIABLE xstep  VARIABLE ystep   ( plaatsverandering )
 2 : xystep xstep @ x +!  ystep @ y +! ;
 3 : xkeer xstep @ NEGATE xstep ! ; ( andere x-richting )
 4 : ykeer ystep @ NEGATE ystep ! ; ( andere y-richting )
 5 : xlinks  x @    0<  IF xkeer THEN ;  ( randcontrole )
 6 : xrechts x @ xmax > IF xkeer THEN ;
 7 : yboven  y @    0<  IF ykeer THEN ;
 8 : yonder  y @ ymax > IF ykeer THEN ;
 9 : clrs    12 EMIT ;    ( scherm schoon ) 192 CONSTANT "bal"
10       ( test balbeweging over scherm )
11 : patroon 0 x ! 0 y ! 1 xstep ! 1 ystep !   ( bal-positie )
12       clrs 0 DO
13                  "bal" xyplot    ( plot de bal )
14                  xystep xlinks xrechts yboven yonder
15              LOOP ;                    ( n → )

112 LIST
 0 ( Woordenschat voor beeldschermspel; bat behandeling )
 1 VARIABLE bat  breedte 2 / bat !  ( begin-positie bat )
 2 : plotbat DUP bat @ 1- ymax plot ( op onderste regel )
 3           DUP bat @    ymax plot
 4               bat @ 1+ ymax plot ;
 5 : bat+1   bat @ xmax 1- LITERAL < IF 1 bat +! THEN ;
 6 : bat-1   bat @                 1 > IF -1 bat +! THEN ;
 7 : hupbat  DUP 122 = IF DROP bat-1 ELSE  ( "z"=batlinks )
 8           DUP  47 = IF DROP bat+1 ELSE  ( "/"=batrechts )
 9                        ABORT THEN THEN ;  ( anders stop )
10 32 CONSTANT "blank" 163 CONSTANT "bat"
11 : zetbat  ?TERMINAL ?DUP
12           IF "blank" plotbat hupbat "bat" plotbat
13           THEN ;
14       ( test bat behandeling )
15 : testbat clrs "bat" plotbat BEGIN zetbat 0 UNTIL ;

113 LIST
 0 ( Woordenschat voor beeldschermspel; squash voorbereiding )
 1 VARIABLE rnd  1234 rnd !    ( beginwaarde voor aselecte rij )
 2 : random rnd @ 1021 * 41 + DUP rnd ! ;          (   → n )
 3 : >rand random U* SWAP DROP ;                   (   → n )
 4       ( xstep wordt -2, -1, 1 of 2 )
 5 : rxstep  4 >rand 1- DUP 0= NOT IF 1- THEN xstep ! ;
 6       ( serveer nieuwe bal )
 7 : nwbal  64 >rand x !  0 y !     ( beginwaarde x en y )
 8           2 >rand 1+ ystep ! rxstep ; ( xstep, ystep )
 9       ( bal scheef terugkaatsen )
10 : wijkaf  2 >rand 2- ystep ! rxstep ;
11
```

```
12 : hitmid  bat @ x @ = ;                                  ( → b)
13 : hitlr   bat @ 1- x @ =  bat @ 1+ x @ = OR ;      ( → b)
14 VARIABLE vlug  300 vlug !   VARIABLE score  0 score !
15 : printscore 13 EMIT ." Score - "   score @ 5 .R ;

114 LIST
 0 ( Woordenschat voor beeldschermspel; solo squash )
 1 : squash
 2      clrs 0 score ! printscore      ( scherm in het begin )
 3      nwbal                          ( serveer een bal )
 4      BEGIN
 5         "bal xyplot  "bat" plotbat ( plot bal en bat )
 6         vlug @ 0 DO zetbat LOOP    ( vertraging voor bat )
 7         "blank" xyplot xystep xlinks xrechts yboven
 8         y @ ymax >                 ( bal op onderste regel? )
 9              IF hitmid IF 10 score +! ykeer ELSE
10                 hitlr  IF  5 score +! wijkaf ELSE
11                           -5 score +! nwbal
12                        THEN THEN
13                 printscore
14              THEN
15    0 UNTIL ;                ( lus tot einde via zetbat )
```

LITERATUUR

1. Moore, C.H. and Rather, E.D., "The FORTH program for spectral line observing", Proc. IEEE, Vol. 61, September 1973.
2. Moore, C.H. and Rather, E.D., "FORTH: A new way to program a mini-computer", Astron. Astrophys. suppl. 15, 1974.
3. James, J.S., "FORTH on microcomputers", Dr. Dobbs, no 26.
4. Rather, E., Brodie, L., Rosenberg, C., "Using FORTH, FORTH-79 standard edition", FORTH Inc., 1979.
5. Moore, C.H., "The Evolution of FORTH, an unusual language", Byte, Vol. 5, August 1980. In dit tijdschriftnummer staan nog enkele andere lezenswaardige artikelen over FORTH.
6. The FORTH Standards Team, "FORTH-79", 1980, distributed by the FORTH Interest Group, P.O. Box 1105, San Carlos, CA 94070, USA.
7. Fritzon, R., "Write your own pseudo-FORTH compiler", Micro-computing, February 1981, p. 76-92 en March 1981, p. 44-57.
8. Loeliger, R.G., "Threaded interpretive languages", Byte Books, 1981.
9. Katzan, H., "Invitation to FORTH", Petrocelli, 1981.
10. Brodie, L., "Starting FORTH", FORTH Inc., 1981; Prentice Hall, 1982.
11. Knecht, K., "Introduction to FORTH, Prentice-Hall, 1982.
12. Hogan, Th., "Discover FORTH: learning and programming the FORTH language", Osborne/Mc Graw-Hill, 1982
13. Voor de programmatuur raadplege men de advertenties van de Mountain View Press Inc. (P.O. Box 4656 Mountain View, CA 94040, USA) in o.a. het tijdschrift Byte.
14. Scanlon, L.J., "FORTH Programming", Prentice-Hall, 1983.
15. Husband, D., "Advanced FORTH", Sigma Technical Press, 1983.
16. Journal of FORTH Application and Research, Vol. 1, September 1983 (Inst. for Applied FORTH Research; 70 Elmwood Ave, Rochester, N.Y. 14611, USA).

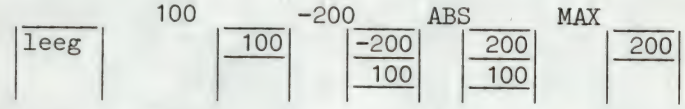
ANTWOORDEN VOOR DE OEFENINGEN IN HOOFDSTUK 1 - 5

Hoofdstuk 1

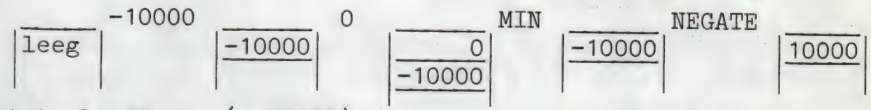
1.
```
1  2  +  3  4  -  *
10  100  9  /  +  5  +
2  3  4  5  6  +  *  *  *
```

2.
```
(20+10)/(20-10)
1+2+3+4
20-(1*2)
```

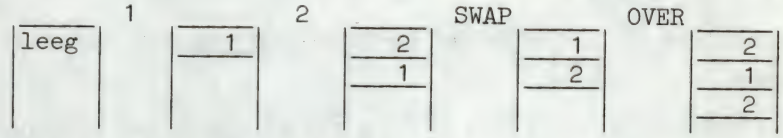
3.



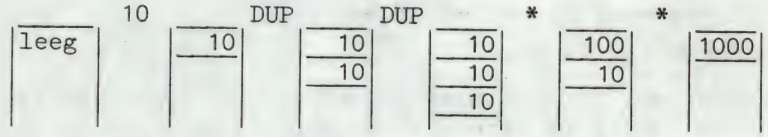
totaal effect: (→ 200)



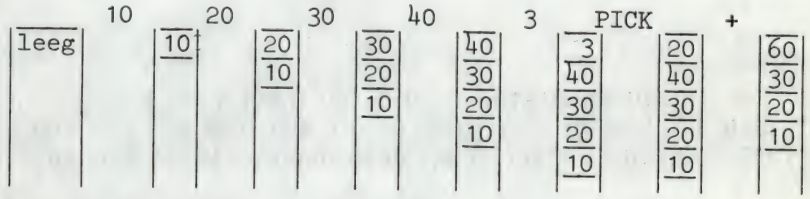
totaal effect: (→ 10000)



totaal effect: (→ 2 1 2)

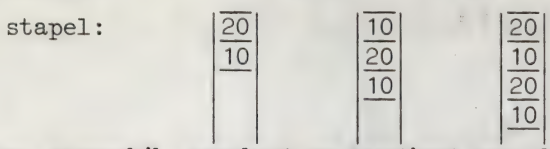


totaal effect: (→ 1000)



totaal effect: (→ 10 20 30 60)

4. Een goede manier om de twee bovenste getallen op de stapel te dupliceren berust op het gebruik van {OVER OVER}, bijvoorbeeld:



Som, verschil, product en quotient van dezelfde twee getallen kan dus als volgt worden berekend:

```
10 20 ok
OVER OVER + . 30 ok
OVER OVER - . -10 ok
OVER OVER * . 200 ok
OVER OVER / . 0 ok
```

Merk ten aanzien van het laatste resultaat op dat de geheeltallige deling 10/20 oplevert: 0 met een rest van 10.

Hoofdstuk 2

1.
```
10 CONSTANT tien
tien 4 * 1 +  CONSTANT fred
```

2.
```
VARIABLE XYZ          -100 XYZ !
VARIABLE A            XYZ @ fred - A !
```

3. De eerste oplossing waaraan je zult denken is:

```
1 X @ + X @ X @ * + X !
```

maar een kortere manier voor het kwadrateren van een variabele is:

```
X @ DUP *
```

en verder gebruik makend van {+!} en verwisseling van optellingen:

```
X @ DUP * 1 + X +!
```

hetgeen een eleganter en sneller oplossing is.

4.
```
a x @ DUP * *   b x @ * + c +
```

5. Als je computer een uitvoerapparaat heeft waarin bijvoorbeeld op geheugenplaats 1000 naar verwezen wordt dan kun je na:

```
1000 CONSTANT apparaat ok
```

met

```
1 apparaat ! ok
```

uitvoer naar dit apparaat sturen, dus net doen alsof het apparaat een variabele is.. Pas op ..., probeer dit zelf niet met geheugenlokatie 1000; misschien staat er op deze plaats iets heel belangrijks!

Hoofdstuk 3

1. `: drie 3 * ;`

   Een snellere oplossing (omdat een optelling bij veel machines veel vlugger gaat dan een vermenigvuldiging), die echter meer ruimte in het Woordenboek vergt, is de volgende:

```
: drie DUP DUP + + ;
```

2.
```
: par 10 EMIT 10 EMIT 13 EMIT ( 2 blanco regels en wagen terug )
     ." Paragraaf "  . ;
```

3.
```
CREATE array -10 , 1 , 10 , 1000 ,
: array 2 * array + ;
```

   Merk op dat we door het woord voor de adresberekening dezelfde naam te geven als het array, als het ware het array 'verbergen' omdat zijn elementen alleen via de adresberekening te bereiken zijn.

4.
```
: dubbelarray
     0 array @  2 *  0 array !
     1 array @  2 *  1 array !
     2 array @  2 *  2 array !
     3 array @  2 *  3 array !
```

   Met een DO-lus (zie hoofdstuk 5) kan het compacter worden opgeschreven:

```
: dubbelarray
     4 0 DO I array @ 2 *  I array ! LOOP ;
```

5.

woord	*effect op stapel*	*commentaar*
DUP	(n1 n2 → n1 n2 n2)	dupliceer getal op stapel
*	(n1 n2 n2 → n1 n3)	n3 = n2 * n2
SWAP	(n1 n3 → n3 n1)	verwissel 2 topwaarden
DUP	(n3 n1 → n3 n1 n1)	dupliceer getal op stapel
*	(n3 n1 n1 → n3 n4)	n4 = n1 * n1
+	(n3 n4 → n5)	n5 = n3 + n4

   Het uiteindelijke effect kan als volgt worden beschreven:

voorbeeld	(n1 n2 → n5)	n5 = n1 * n1 + n2 * n2

Hoofdstuk 4

1.

woord	*effect op stapel*	*commentaar*
1	(→ 1)	1 op de stapel
2	(1 → 1 2)	2 er boven op
>	(1 2 → 0)	1<2, dus 'false' op de stapel
-4	(→ -4)	-4 op de stapel
0<	(-4 → 1)	-4<0, dus 'true' op de stapel
5	(→ 5)	5 op de stapel
0>	(5 → 1)	5 > 0, dus 'true' op de stapel
NOT	(1 → 0)	en nu 'false' op de stapel

2.
```
: teken DUP 0> IF ." positief"
                ELSE DUP 0= IF ." nul"
                               ELSE ." negatief"
                               THEN
                TEN ;       ( getal blijft op stapel )
```

3.
```
1101101 XOR 1010001 = 0111100
1010 OR 101 = 1111
```

woord	*effect stapel*	*commentaar*
4	(→ 4)	4 op de stapel
5	(4 → 4 5)	5 er boven op
=	(4 5 → 0)	topwaarden ongelijk, dus 'false'
2	(0 → 0 2)	2 er boven op
3	(0 2 → 0 2 3)	3 weer daarboven
<	(0 2 3 → 0 1)	2<3, dus 'true' op de stapel
OR	(0 1 → 1)	resultaat is 'true'

4.
```
A @ 2 = B @ 2 = AND NOT IF 4 A ! THEN
```

5. {0=} en {NOT} **hebben hetzelfde effect voor logische waarden, maar niet voor getallen!**

6.

woord	*effect stapel*	*commentaar*
OVER	(n1 n2 → n1 n2 n1)	
OVER	(n1 n2 n1 → n1 n2 n1 n2)	n1 en n2 gedupliceerd
>	(n1 n2 n1 n2 → n1 n2 b)	b=true als n1>n2
IF	(n1 n2 b → n1 n2)	alleen SWAP als n1>n2
SWAP	(n1 n2 → n2 n1)	
THEN		
DROP	(→ n2 of n1)	n2 als n1>n2, anders n1

Het effect is dat de kleinste waarde van n1 en n2 op de stapel achterblijft, m.a.w. hetzelfde effect als dat van {MIN}.

woord	*effect stapel*	*commentaar*
DUP	(n → n n)	dupliceer
IF	(n n → n)	DUP alleen als n≠0
DUP	(n → n n)	dupliceer
THEN		

Het effect is dat het getal op de stapel alleen gedupliceerd wordt wanneer het niet-nul is, m.a.w. hetzelfde effect als dat van {?DUP}.

Hoofdstuk 5

1.
```
: sterren CR                        ( nieuwe regel )
          DUP 0 DO                  ( buitenlus )
                  0 DO              ( binnenlus )
                      ." *"         ( druk * af )
                    LOOP
                    CR              ( nieuwe regel )
                LOOP ;
```

2.
```
: sigma 0                          ( accumulator = 0 )
        SWAP 1+                    ( bovengrens + 1 )
        ROT                        ( index boven op stapel )
        DO
           I +
        LOOP ;                     ( laat som op stapel )
```

3.
```
: vertraag 1000 0 DO LOOP ;        ( vertraging ± 1 seconde )
: aftellen 0 SWAP                  ( verwissel index en limiet )
           DO
             I .                   ( druk waarde af )
             vertraag
       -1    +LOOP
             ." We zijn gestart" ;
```

4.
```
vb1 0 3 6 9 12 15 ok
vb2 10 9 8 7 6 5 4 3 2 1 0 ok
vb3 5 10 15 20 25 ....95 100 ok
```

5. **Besluit eerst hoe je de parameters zult invoeren, bijvoorbeeld:**

```
3 0 20 deelbaar
```

om alle getallen in het bereik 0 t/m 20, die precies deelbaar zijn door 3, af te drukken.

```
: deelbaar
    ROT ROT                        ( index en limiet naar de top )
    SWAP 1+ SWAP                   ( limiet + 1 )
    DO
      DUP                          ( dupliceer deler )
      I                            ( te onderzoeken getal )
      SWAP MOD                     ( bepaal de rest )
      0= IF                        ( als rest 0 is )
           I                       ( getal afdrukken )
         THEN
    LOOP DRP ;                     ( stapel schoon maken )
```

```
3 0 20 deelbaar 3 6 9 12 15 18 ok
```

6.
```
: dump BEGIN
          CR
          8 0 DO                   ( lus voor 8 regels )
               DUP 6 .R            ( druk adres af )
               8 0 DO              ( lus voor 8 bytes )
                     DUP C@ 3 .R SPACE 1+
                 LOOP
               CR
              LOOP
       KEY 32 -                    ( toets aangeslagen ? )
       UNTIL                       ( lus in voor een spatie )
       DROP ;                      ( anders klaar )
```

VERKLARENDE WOORDENLIJST (MET ASCII-TABEL)

Woorden die in het voorgaande gebruikt, maar niet altijd verklaard zijn, worden hier van een toelichting voorzien.

Adres
Een 16 bits waarde die de plaats van een byte in het geheugen geeft.

ASCII
Dit is de afkorting van 'American Standard Code for Information Interchange', de code voor de voorstelling van karakters door een combinatie van 7 bits. In de volgende tabel staan deze karakters met hun decimaal en hexadecimaal geschreven bit-combinaties. De meeste computerfabrikanten gebruiken deze code.

Kar	*Dec*	*Hex*	*Kar*	*Dec*	*Hex*	*Kar*	*Dec*	*Hex*	*Kar*	*Dec*	*Hex*
NUL	0	00	SPACE	32	20	@	64	40	'	96	60
SOH	1	01	!	33	21	A	65	41	a	97	61
STX	2	02	"	34	22	B	66	42	b	98	62
ETX	3	03	#	35	23	C	67	43	c	99	63
EOT	4	04	$	36	24	D	68	44	d	100	64
ENQ	5	05	%	37	25	E	69	45	e	101	65
ACK	6	06	&	38	26	F	70	46	f	102	66
BEL	7	07	'	39	27	G	71	47	g	103	67
BS	8	08	(	40	28	H	72	48	h	104	68
HT	9	09	)	41	29	I	73	49	i	105	69
LF	10	0A	*	42	2A	J	74	4A	j	106	6A
VT	11	0B	+	43	2B	K	75	4B	k	107	6B
FF	12	0C	,	44	2C	L	76	4C	l	108	6C
CR	13	0D	-	45	2D	M	77	4D	m	109	6D
SO	14	0E	.	46	2E	N	78	4E	n	110	6E
SI	15	0F	/	47	2F	O	79	4F	o	111	6F
DLE	16	10	0	48	30	P	80	50	p	112	70
DC1	17	11	1	49	31	Q	81	51	q	113	71
DC2	18	12	2	50	32	R	82	52	r	114	72
DC3	19	13	3	51	33	S	83	53	s	115	73
DC4	20	14	4	52	34	T	84	54	t	116	74
NAK	21	15	5	53	35	U	85	55	u	117	75
SYN	22	16	6	54	36	V	86	56	v	118	76
ETB	23	17	7	55	37	W	87	57	w	119	77
CAN	24	18	8	56	38	X	88	58	x	120	78
EM	25	19	9	57	39	Y	89	59	y	121	79
SUB	26	1A	:	58	3A	Z	90	5A	z	122	7A
ESC	27	1B	;	59	3B	[	91	5B	}	123	7B
FS	28	1C	<	60	3C	\	92	5C	|	124	7C
GS	29	1D	=	61	3D	]	93	5D	{	125	7D
RS	30	1E	>	62	3E	↑	94	5E	~	126	7E
US	31	1F	?	63	3F	_	95	5F	DEL	127	7F

De karakters in de linkerkolom (en DEL) veroorzaken niet een afdruk, maar 'besturen' o.a. de randapparatuur. Enkele bekende zijn:

```
BS = 'backspace' één karakter (wagen 1 positie terug)
LF = 'line feed' (nieuwe regel)
FF = 'form feed' (nieuwe pagina of scherm schoon)
CR = 'carriage return' (begin een regel vooraan)
```

Assembleertaal
Een assembleertaal is de symbolische schrijfwijze voor een machinetaal en daarom erg afhankelijk van de door een computerfabrikant gekozen instructiecode. Dit is dus in tegenstelling tot talen als BASIC of FORTH, die op verschillende computertypen bruikbaar zijn. In uitgebreide FORTH systemen is het mogelijk om instructies in een assemblertaal te gebruiken in een FORTH programma. Definities die dit mogelijk maken, heten CODE-definities; zij hebben ongeveer dezelfde structuur als onze colon-definities. Voor een machine met een 8080 microprocessor kan een CODE-definitie om een getal op de stapel te verdubbelen er bijvoorbeeld als volgt uitzien:

```
CODE  DUBBEL              ( nieuwe definitie, genoemd DUBBEL )
        POPHL CALL        ( stapeltop naar HL )
        H     DAD         ( tel HL bij zichzelf op )
        PUSHHL JMP        ( zet HL op stapel )
END-CODE
```

Merk op dat in 8080 instructies eerst de operand wordt geschreven en daarachter de operatiecode. Met code-definities kunnen kritische delen in een programma versneld worden, maar het nadeel is dat de programmeur de assemblertaal van een microprocessor moet kennen en dat het vervaardigde programma niet op andere systemen kan worden verwerkt.

Binair
Een ander woord voor twee-tallig. De decimale getallen 0, 1, 2, 3, ... zijn binair geschreven: 0, 1, 10, 11, 100, 101, ...

Boolean
Een ander woord voor een 'logische' variabele of constante, die slechts een van de twee 'logische' waarden 'false' of 'true' kunnen aannemen. In FORTH kan ieder 16 bits getal als een Boolean opgevat worden, waarbij 16 nullen 'false' voorstellen en iedere andere combinatie van nullen en enen 'true'.

Byte
Een 8 bits waarde. Daar FORTH 16 bits getallen op zijn stapel heeft, worden daar byte-waarden voorgesteld als 16 bits getallen, waarvan de eerste 8 bits nullen zijn.

Definiërend woord
Een woord dat bij uitvoering een toevoeging aan het Woordenboek veroorzaakt van het daarop volgende woord in de invoersliert. Voorbeelden van definiërende woorden zijn {:}, {CREATE}, {VARIABLE}, {CONSTANT} en {VOCABULARY}.

Getallen
FORTH heeft bewerkingen voor de volgende getal-'typen':

type	
bit	0 .. 1
karakter	0 .. 127
byte	0 .. 255
integer (n)	-32768 .. 32767
getal zonder teken (un)	0 .. 65535
dubbele nauwkeurigheid getal (d)	-2147483648 .. 2147483647
dubbele nauwkeurigheid getal zonder teken (ud)	0 .. 4294967295

(de afkortingen tussen haakjes worden gebruikt in de stapel-notatie) Dubbele nauwkeurigheid getallen worden op de stapel voorgesteld met twee opeenvolgende 16 bits waarden met bovenop de eerste 16 van de 32 bits. Alle andere getal-typen worden met 16 bits waarden voorgesteld , met nullen voorop voor karakters of bytes.

Hexadecimaal
Een getalvoorstelling in het 16-tallig stelsel, zodat de getallen 0 t/m 16 geschreven worden als 0, 1, ... 9, A, B, C, D, E, F, 10 (zie ook bij ASCII).

Infix
Een notatie-afspraak voor uitdrukkingen waarin de (bewerkings-) operatoren *tussen* de grootheden, waarop ze werken, worden geplaatst (zoals we altijd gewend zijn); bijvoorbeeld voor deling van 10 door 3: 10/3.

Invoersliert
Een rij karakters die geïnterpreteerd moet worden. Deze rij kan afkomstig zijn van het toetsenbord (via de invoerbuffer) of van schijf of cassette (via een blokbuffer). De waarden van de variabelen {>IN} en {BLK} bepalen wat de actuele invoersliert is.

Karakter
Een verzamelnaam voor letters, cijfers, leestekens en andere symbolen, die op papier of beeldscherm zichtbaar kunnen worden gemaakt, alsmede signalen voor computer randapparatuur (zie ook bij ASCII).

Literal
In FORTH is een literal een in een colon-definitie voorkomend getal, dat alleen dat getal zelf voorstelt.

Postfix
Een notatie-afspraak voor uitdrukkingen waarin in tegenstelling tot de infix-notatie de (bewerkings-)operatoren geplaatst worden *achter* de grootheden waarop ze werken; bijvoorbeeld 10 3 /. Postfix uitdrukkingen (waarin nooit haakjes nodig zijn) kunnen met behulp van een stapel direct verwerkt worden. In FORTH moeten alle reken- en ook logische bewerkingen in de postfix-notatie worden geschreven.

String
Een rij byte-waarden in het geheugen die een tekst kan voorstellen.

Twee-complement representatie
Een representatiewijze voor getallen waarbij de eerste bit het teken van het getal aangeeft (0 voor niet negatieve getallen en 1 voor negatieve getallen) en waarbij de representatie van een negatief getal wordt verkregen door het corresponderende positieve getal af te trekken van 0. Voor 16 bits getallen krijgen we zo:

binair	*decimaal*
0111111111111111	32767, het grootste positieve getal
0000000000000000	0, het kleinste positieve getal
1111111111111111	-1, het grootste negatieve getal
1000000000000000	-32768, het kleinste negatieve getal

Vaste komma getal
Een methode voor het representeren van decimale ('reële') getallen door een vaste plaats aan te nemen voor de decimaal-punt. Als bijvoorbeeld de decimaal-punt twee plaatsen naar links staat dan stelt een 1 op de stapel het 'vaste komma getal' 0.01 voor, en 1234 op de stapel het getal 12.34.

Vertaalwoord (compilatie-woord)
Een woord, dat opgenomen in een colon-definitie, zowel tijdens de vertaalfase als tijdens de verwerkingsfase een bepaalde uitwerking heeft. Vertaalwoorden zijn o.a. {IF},{ELSE}, {THEN}, {DO}, {LOOP}, {+LOOP}, {BEGIN}, {UNTIL}, {WHILE} en {LITERAL}.

Woord
In de FORTH terminologie is een woord iedere aan weerskanten door een spatie begrensde rij van karakters in een invoersliert. In de gebruikelijke computer terminologie is een woord daarentegen (de inhoud van) een geheugencel van bepaalde grootte (uitgedrukt in bits). In FORTH wordt een grootheid van 16 bits altijd een GETAL, ADRES of soms CEL genoemd.

Woordenboek
Een gegevensstructuur ('dictionary') waarin de definities van alle woorden, zowel die van het systeem als die toegevoegd door de programmeur, voorkomen. Door de naam van een woord te zoeken, is een gewenst woord te vinden.

Woordenschat
Een deelverzameling van het Woordenboek met een eigen naam ('vocabulary'). Meer dan één woordenschat kan in het Woordenboek voorkomen en iedere woordenschat is verbonden met de primaire FORTH woordenschat.

REGISTER

Dit register omvat de verzameling van FORTH-79 standaardwoorden met een verwijzing naar de pagina waar de formele beschrijving van een woord staat. Een informele beschrijving met voorbeelden is als regel in de daaraan voorafgaande pagina's te vinden.

ACADEMIC SERVICE INFORMATICA UITGAVEN

INLEIDINGEN

Computers en onze samenleving van M.A. Arbib

Basiskennis informatieverwerking van Jan Everink

De viewdata revolutie van S. Fedida en R. Malik

De informatiemaatschappij van Jan Everink

Informatica, een theoretische inleiding van dr. L.P.J. Groenewegen en prof.dr. A. Ollongren

AIV, Automatisering van de informatieverzorging van ir. Th.J. Derksen, drs. H.W. Crins en drs. L.B. Essink

Organisatie, informatie en computers van David M. Kroenke

MICROCOMPUTERS

Programmeercursus Microsoft BASIC van Nok van Veen

Werken met bestanden in BASIC van L. Finkel en J.R. Brown

Werken met bestanden in Apple-BASIC van L. Finkel en J.R. Brown

Werken met Visicalc van C. Klitzner en M.J. Plociak, Jr.

CP/M: een gids voor zelfstudie van J.N. Fernandez en R. Ashley

Cursus Z-80 assembleertaal van Roger Hutty

Exidy sorcerer en BASIC van Nok van Veen e.a.

TRS-80 BASIC: een gids voor zelfstudie van B. Albrecht e.a.

TRS-80 BASIC voor gevorderden van Don Inman e.a.

Ontdek de ZX-Spectrum van Tim Hartnell

PROGRAMMEREN/PROGRAMMEERTALEN

Inleiding tot het programmeren, deel 1 van ir. J.J. van Amstel e.a.

Inleiding tot het programmeren, deel 2 van ir. J.J. van Amstel e.a.

JSP-Jackson struktureel programmeren van Henk Jansen

Aspecten van programmeertalen van ir. J.J. van Amstel en ir. J.A.A.M. Poirters

Programmeertalen, een inleiding van ir. J.J. van Amstel e.a.

Het Groot Pascal Spreukenboek van H.F. Ledgard , P.A. Nagin en J.F. Hueras

Cursus Pascal van prof.dr. A. van der Sluis en drs. C.A.C. Görts

Cursus eenvoudig Pascal van prof.dr. A. van der Sluis en C.A.C. Görts

Inleiding programmeren in Pascal van C. van de Wijgaart

BASIC, EIT-serie, deel 3

Cursus BASIC van ir. R. Bloothoofd e.a.

Cursus COBOL van dr. A. Parkin

Struktuur en stijl in COBOL van ir. E. Dürr en dr.ir. F. Mulder

Cursus FORTRAN 77 van J.N.P. Hume en R.C. Holt

Programmeren in LISP van prof.dr. L.L. Steels

Cursus ALGOL 60 van prof.dr. A. van der Sluis en drs. C.A.C. Görts

Programmeren, deel 2: Van analyse tot algoritme van prof.drs. C. Bron

Inleiding programmeren en programmeertechnieken, EIT-serie, deel 1

Inleiding programmeren van prof.dr. R.J. Lunbeck

SYSTEEMPROGRAMMATUUR

Bedrijfssystemen, EIT-serie, deel 4

Systeemprogrammatuur van drs. H. Alblas

Vertalerbouw van drs. H. Alblas e.a.

BESTANDSORGANISATIE/DATABASES

Informatiestructuren, bestandsorganisatie en bestandsontwerp, EIT-serie, deel 5

Gegevensstructuren van R. Engmann e.a.

Bestandsorganisatie van prof.dr. R.J. Lunbeck en drs. F. Remmen

Databases van drs. F. Remmen

INFORMATIEANALYSE/SYSTEEMONTWERP

Voorbereiding van computertoepassingen van prof.dr. A.B. Frielink

Simulatie, een moderne methode van onderzoek van drs. S.K.T. Boersma en ir. T. Hoenderkamp

Systeemontwikkeling volgens S.D.M. van H.B. Eilers

Een samenvatting van de System Development Methodology SDM van PANDATA

Cases op het gebied van administratieve organisatie en informatieverzorging (inclusief systeemontwerp) van prof.dr. P.G. Bosch en H.A. te Rijdt

Uitwerkingenboek bij Cases van prof.dr. P.G. Bosch en H.A. te Rijdt

Gegevensanalyse van R.P. Langerhorst

Evaluation of methods and techniques for the analysis, design and implementation of information systems, editors: J. Blank en M.J. Krijger

Analyse van informatiebehoeften en de inhoudsbeschrijving van een databank van prof.dr. P.G. Bosch en ir. H.M. Heemskerk

Eerlijk en helder van prof.dr. P.G. Bosch

THEORETISCH/COMPUTERSCHAAK/TOEPASSINGEN

Abstracte automaten en grammatica's van prof.dr. A. Ollongren en ir. Th.P. van der Weide

De tekstmachine van dr. M. Boot en drs. H. Koppelaar

Computerschaak, schaakwereld en kunstmatige intelligentie van dr. H.J. van den Herik

Lineaire programmering als hulpmiddel bij de besluitvorming van dr. S.W. Douma

Simulatie en sociale systemen, redaktie: J.L.A. Geurts en J.H.L. Oud

Onderneming en overheid in systeem-dynamisch perspectief, redaktie: A.F.G. Hanken en J.H.L. Oud

INFORMATIE OVER DEZE PUBLIKATIES BIJ:

Academic Service, Postbus 96996, 2509 JJ Den Haag
Tel.: 070-247238

FORTH-79 OVERZICHTSKAART

Stapel notatie: (gewone stapel eerst → gewone stapel daarna)

Operanden:

n, n1, ...	16 bits waarde	
d, d1, ...	32 bits waarde	
adr	16 bits adres	
byte	16 bits waarde, waarvan alleen de laatste 8 bits van belang zijn voor een bewerking	
kar	16 bits waarde, waarvan de laatste 7 bits een ASCII-karakter representeren	
b	16 bits logische waarde, 'false' als alle bits nul zijn en anders 'true'	
u....	geeft een getal zonder teken aan	

Stapel manipulatie

DUP	(n → n n)	Dupliceer getal boven op stapel
DROP	(n →)	Verwijder getal boven op stapel
SWAP	(n1 n2 → n2 n1)	Verwissel 2 bovenste getallen
OVER	(n1 n2 → n1 n2 n1)	Dupliceer 2-de getal op stapel
ROT	(n n1 n2 → n1 n2 n)	Roteer 3-de getal naar boven
PICK	(n1 → n2)	Dupliceer n1-de getal van stapel
ROLL	(n →)	Roteer n-de getal naar boven
?DUP	(n → n (n))	Dupliceer als n ≠ 0
DEPTH	(→ n)	Tel aantal gegevens op de stapel
>R	(n →)	Bovenste gegeven naar terugkeer-stapel
R>	(→ n)	Van terugkeer- naar gewone stapel
R@	(→ n)	Kopieer top van terugkeer-stapel

Vergelijkingsbewerkingen

<	(n1 n2 → b)	True als n1 kleiner is dan n2
=	(n1 n2 → b)	True als n1 gelijk is aan n2
>	(n1 n2 → b)	True als n1 groter is dan n2
0<	(n → b)	True als n negatief is
0=	(n → b)	True als n nul is
0>	(n → b)	True als n groter is dan 0
D<	(d1 d2 → b)	True als d1 kleiner is dan d2
U<	(un1 un2 → b)	Vergelijk ze zonder teken
NOT	(b → -b)	Verander logische waarde

Invoer en uitvoer van karakters

CR	(→)	Wagen terug en nieuwe regel
EMIT	(kar →)	Druk karakter af
SPACE	(→)	Druk een spatie af
SPACES	(n →)	Druk n spaties af
." tekst"	(→)	Druk de tekst afgesloten met " af
TYPE	(adr n →)	Druk n karakters vanaf adr af
COUNT	(adr → adr+1 n)	n=aantal karakters vanaf adr+1
-TRAILING	(adr n1 → adr n2)	Verlaag n1 met aantal spaties achteraan
KEY	(→ kar)	Lees een karakter van het toetsenbord
EXPECT	(adr n →)	Zet n karakters (of tot return) van toetsenbord in geheugen vanaf adr
QUERY	(→)	Zet 80 karakters (of tot return) van toetsenbord in invoerbuffer
WORD	(kar → adr)	Lees van invoersliert tot kar en zet aantal karakters op adr in buffer

Invoer en uitvoer van getallen

BASE	(→ adr)	Systeem-variabele voor radix
DECIMAL	(→)	Maak de radix 10
.	(n →)	Druk n af en dan een spatie
U.	(un →)	Als vorige voor getal zonder teken
CONVERT	(d1 adr1 → d2 adr2)	Zet string op adr1 om in dubb.nauwk. getal en tel er d1 bij op voor d2
<#	(→)	Begin van opmaak van een getal
#	(ud1 → ud2)	Zet volgende cijfer van ud1 om; HOLD
#S	(ud → 0 0)	Zet resterende significante cijfers van ud om en HOLD ze allemaal
HOLD	(kar →)	Zet karakter in opgemaakte string
SIGN	(n ud → ud)	HOLD min-teken voor negatieve n
#>	(ud → adr n)	Einde opmaak van getal voor TYPE

Invoer en uitvoer voor achtergrondgeheugen

LIST	(n →)	Lijst blok n af en n naar SCR
LOAD	(n →)	Interpreteer blok n; dan gewoon door
SCR	(→ adr)	Systeem-variabele die nummer van afgelijst blok bevat
BLOCK	(n → adr)	Adres in geheugen van blok n, nadat dit zo nodig gelezen is
UPDATE	(→)	Kenmerk 'gewijzigd' in laatste blok
BUFFER	(n → adr)	Ken buffer op adr toe aan blok n
SAVE-BUFFERS	(→)	Alle gewijzigde buffers redden
EMPTY-BUFFERS	(→)	Kenmerk 'leeg' voor alle buffers

Definiërende woorden

: <naam>	(→)	Begin colon-definitie van <naam>
;	(→)	Einde van colon-definitie
VARIABLE <naam>	(→)	Definitie van variabele <naam>
<naam>	(→ adr)	Adres van variabele op de stapel
CONSTANT <naam>	(n →)	Definitie van constante <naam> = n
<naam>	(→ n)	Waarde van constante op de stapel
VOCABULARY <naam>	(→)	Definitie van woordenschat <naam>, wordt CONTEXT woordenschat bij uitvoering
CREATE <naam>	(→)	Maakt lege toevoeging aan woordenschat
<naam>	(→ adr)	Geeft adres parameterveld
DOES>	(→ adr)	Voor definiëren van nieuwe def.woorden

Reken- en logische bewerkingen

+	(n1 n2 → som)	Optelling
-	(n1 n2 → verschil)	Aftrekking (n1 - n2)
*	(n1 n2 → product)	Vermenigvuldiging
/	(n1 n2 → quotient)	Deling (n1/n2 met afronding naar 0)
MOD	(n1 n2 → rest)	Rest van n1/n2 met teken van n1
/MOD	(n1 n2 → rest quot)	Deling met rest en quotient
1+	(n → n+1)	Optelling van 1
1-	(n → n-1)	Aftrekking van 1
2+	(n → n+2)	Optelling van 2
2-	(n → n-2)	Aftrekking van 2
D+	(d1 d2 → dsom)	Dubbele nauwkeurigheid optelling
*/	(n1 n2 n3 → quot)	n1*n2/n3 met dubbele nauwk.product
*/MOD	(n1 n2 n3 → rest q.)	Zoals */ met bovendien rest
U*	(un1 un2 → ud)	Dubb.nauwk. product (zonder teken)
U/MOD	(ud un → urest uq.)	Dubb.nauwk. getal/getal (")
MAX	(n1 n2 → max)	Grootste van n1 en n2
MIN	(n1 n2 → min)	Kleinste van n1 en n2
ABS	(n → |n|)	Absolute waarde
NEGATE	(n → -n)	Tekenomkeer (2-complement)
DNEGATE	(d → -d)	Tekenomkeer van dubb.nauwk. getal
AND	(n1 n2 → and)	AND voor 2 bitrijen
OR	(n1 n2 → or)	OR voor 2 bitrijen
XOR	(n1 n2 → xor)	XOR voor 2 bitrijen

Geheugenbewerkingen

@	(adr → n)	Vervang adres door getal op adres
!	(n adr →)	Bewaar n op adres
C@	(adr → byte)	Vervang adres door byte op adres
C!	(byte adr →)	Bewaar byte op adres
?	(adr →)	Druk getal op adres af
+!	(n adr →)	Tel n op bij getal op adres
MOVE	(adr1 adr2 n →)	Verplaats n getallen van adr1→adr2
CMOVE	(adr1 adr2 n →)	Verplaats n bytes van adr1→adr2
FILL	(adr n byte →)	Zet n bytes in geheugen vanaf adr

Besturings-structuren

DO	(eind+1 start →)	Doe DO..LOOP of +LOOP
LOOP	(→)	Index+1; luseinde als index>eind
+LOOP	(n →)	Index+n; luseinde als index>eind bij n>0, of index=eind bij n<0
I	(→ n)	Zet momentane indexwaarde op stapel
J	(→ n)	Zet index van buitenlus op stapel
LEAVE	(→)	Bewerkstellig einde van de DO-lus
IF	(b →)	In structuur IF..true..THEN of IF..true..ELSE..false..THEN
ELSE	(→)	
THEN	(→)	Voer true- of false-opdrachten uit, afhankelijk van de logische b
BEGIN	(→)	Begin van WHILE- of UNTIL-lus
UNTIL	(b →)	Lus BEGIN..UNTIL tot b true is
WHILE	(b →)	In constructie BEGIN..WHILE..REPEAT
REPEAT	(→)	lus herhalen zolang b true is
EXIT	(→)	Voortijdig verlaten van colon-def.
EXECUTE	(adr →)	Voer opdracht op adr uit

Woordenboekbeheer

CONTEXT	(→ adr)	Wijzer in het Woordenboek voor het opsporen van woorden
CURRENT	(→ adr)	Wijzer in het Woordenboek voor toevoegingen
FORTH	(→)	Standaard Woordenboek wordt CONTEXT
DEFINITIONS	(→)	CURRENT woordenschat wordt CONTEXT
' <naam>	(→ adr)	Zet adres van parameterveld op stapel
FIND	(→ adr)	Zet code-veld adres van volgende woord in invoersliert op stapel
FORGET <naam>	(→)	Vergeet definities van <naam> en volgende

Vertaler woorden

ALLOT	(n →)	Voeg n bytes toe aan het parameterveld van laatst gedefinieerde woord
IMMEDIATE	(→)	Kenmerk laatst gedefinieerde woord als direct uit te voeren
LITERAL	(n →)	Zet n als literal in het Woordenboek
STATE	(→ adr)	Systeem-variabele ≠0 bij vertaling
[	(→)	Stop vertaling invoersliert en start uitvoering
]	(→)	Stop uitvoering , start vertaling
,	(n →)	Zet n in het Woordenboek
COMPILE	(→)	Adres van code-veld van volgende woord in Woordenboek zetten
[COMPILE]	(→)	Vertaal volgende woord, zelfs als het een onmiddellijk woord is

Diversen

(	(→)	Begin van commentaar, afgesloten met)
HERE	(→ adr)	Adres van vrije plaats in Woordenboek
PAD	(→ adr)	Adres van 64 bytes 'kladregel'
>IN	(→ adr)	Geeft plaats in invoerbuffer aan
BLK	(→ adr)	Geeft nummer van het geïnterpreteerde blok
ABORT	(→)	Alle stapels schoon en start weer met toetsenbord
QUIT	(→)	Als ABORT, maar gewone stapel intact
79-STANDARD	(→)	Verifieer FORTH-79 norm

Het bovenstaande is ontleend aan het werk van het FORTH Standards Team, P.O. Box 1105, San Carlos, CA 94070, USA.

FORTH is oorspronkelijk ontstaan uit de wens om allerlei apparaten, uiteenlopende van spectrometers tot grote radiotelescopen, te kunnen besturen. Van de te gebruiken taal werd geëist dat die zowel op grote als op kleine computers interactief toe te passen zou zijn, maar dat de verwerking van de programma's snel zou zijn. Bestaande programma's voldeden niet aan deze eisen.

Omdat FORTH-systemen niet alleen voor grote computers (als de IBM 360, Univac 1108, Burroughs 5500, CDC 6400) beschikbaar kwamen, maar ook voor middelgrote machines (als PDP 10 en 11 serie, Data General NOVA, enz.) en voor kleine machines (als Apple II en III, Atari, TRS-80 I en III, IBM PC, Osborne, Victor en verder eigenlijk alle microprocessoren met een CP/M Operating System) groeide de populariteit van deze taal snel. Toepassingen van deze taal treft men nu niet alleen aan bij de besturing van apparaten, maar ook bij tekstverwerking en computerspelletjes (waarvan voorbeelden in dit boek).